FORMAL METHODS SPECIFICATION AND ANALYSIS GUIDEBOOK FOR THE VERIFICATION OF SOFTWARE AND COMPUTER SYSTEMS

VOLUME II: A PRACTITIONER'S COMPANION

FOREWORD

This volume presents technical issues involved in applying mathematical techniques known as Formal Methods to specify and analytically verify aerospace and avionics software systems. The first volume in this two-part series, NASA-GB-002-95 [NASA-95a], dealt with planning and technology insertion. This second volume discusses practical techniques and strategies for verifying requirements and high-level designs for software intensive systems. The discussion is illustrated with a realistic example based on NASA's Simplified Aid for EVA (Extravehicular Activity) Rescue [SAFER94a,SAFER94b]. The volume is intended as a "companion" and guide for the novice formal methods and analytical verification practitioner. Together, the two volumes address the recognized need for new technologies and improved techniques to meet the demands inherent in developing increasingly complex and autonomous systems. The support of NASA's Safety and Mission Quality Office for the investigation of formal methods and analytical verification techniques reflects the growing practicality of these approaches for enhancing the quality of aerospace and avionics applications.

Both volumes of the guidebook are electronically available at JPL via the URL http://eis.jpl.nasa.gov/quality/Formal_Methods/. PVS source files for the SAFER example are available on LaRC's Web server in the directory ftp://atb-www.larc.nasa.gov/Guidebooks/.

Major contributors to the guidebook include Judith Crow, lead author (SRI International); Ben Di Vito, SAFER example author (VíGYAN); Robyn Lutz (NASA - JPL); Larry Roberts (Lockheed Martin Space Mission Systems and Services); Martin Feather (NASA - JPL); and John Kelly (NASA - JPL), task lead. Special thanks go to John Rushby (SRI International) who provided valuable material and guidance, Sam Owre (SRI International) who graciously supplied wide-ranging technical expertise, Gerard Holzmann (Lucent Technologies) and Peter Gorm Larsen (IFAD) both of whom gave particularly thorough and thoughtful reviews, and Valerie Mathews (NASA - JPL) who served as guidebook review and publication coordinator. Special acknowledgment is also extended to NASA sponsors Kathryn Kemp (Deputy Director, NASA IV&V Facility), George Sabolish (NASA - Ames), Rick Butler (NASA - Langley), and Ernie Fridge (NASA - Johnson).

This document is a product of NASA's Software Program, an agency-wide program that promotes continual improvement in software engineering and assurance within NASA. The goals and strategies of this program are documented in the NASA Software Strategic Plan [NASA-95b]. Funding for this guidebook was provided by NASA's Office of Safety and Mission Assurance. Additional information about this program and its products is available via the World Wide Web at http://www.ivv.nasa.gov.

Contents

1 Introduction 1

2 The Practical Application of Formal Methods 5
- 2.1 What Are Formal Methods? 5
- 2.2 Roles of Formal Methods 6
- 2.3 Formal Methods: Degree of Formalization and Scope of Use 6
  - 2.3.1 Levels of Formalization 7
  - 2.3.2 Scope of Formal Methods Use 8
- 2.4 Reasonable Expectations for Formal Methods 9
- 2.5 The Method Underlying Formal Methods 10
- 2.6 An Introduction to SAFER 13

3 Requirements 19
- 3.1 Requirements and Formal Methods 20
  - 3.1.1 Impact of Requirements Specification on Formal Methods 20
    - 3.1.1.1 Level of Requirements Capture 20
    - 3.1.1.2 Explicitness of Requirements Statement 20
    - 3.1.1.3 Clarity of Delineation between a System and Its Environment 20
    - 3.1.1.4 Traceability of Requirements 21
    - 3.1.1.5 Availability of Underlying Rationale and Intuition 21
  - 3.1.2 Impact of Formal Methods on Requirements 22
- 3.2 Conventional Approaches to Requirements Validation 23
- 3.3 SAFER Requirements 25

4 Models 27
- 4.1 Mathematical Models 27
  - 4.1.1 Characteristics of Mathematical Models 28
    - 4.1.1.1 Abstraction 28
    - 4.1.1.2 Focus 29
    - 4.1.1.3 Expressiveness Versus Analytic Power 29
    - 4.1.1.4 Intuitive Versus Nonintuitive Representation 29

4.1.1.5 Accuracy 30
4.1.2 Benefits of Mathematical Models 30
4.1.3 Mathematical Models for Discrete and Continuous Domains . . . 31
4.2 Continuous Domain Modeling 32
4.3 Discrete Domain Modeling 33
4.3.1 Functional Models 34
4.3.2 Abstract State Machine Models 36
4.3.3 Automata-Based Models 39
4.3.3.1 $*$-Automata 39
4.3.3.2 ω-Automata 39
4.3.3.3 Timed Automata 40
4.3.3.4 Hybrid Automata 40
4.3.4 Object-Oriented Models 41
4.4 A Model for the SAFER Avionics Controller 46

5 Formal Specification **53**
5.1 Formal Specification Languages 54
5.1.1 Foundations 54
5.1.2 Features 56
5.1.2.1 Explicit Semantics 57
5.1.2.2 Expressiveness 57
5.1.2.3 Programming Language Datatypes and Constructions . 57
5.1.2.4 Convenient Syntax 58
5.1.2.5 Diagrammatic Notation 58
5.1.2.6 Strong Typing 58
5.1.2.7 Total versus Partial Functions 59
5.1.2.8 Refinement 60
5.1.2.9 Introduction of Axioms and Definitions 60
5.1.2.10 Encapsulation Mechanism 62
5.1.2.11 Built-in Model of Computation 63
5.1.2.12 Executability 63
5.1.2.13 Maturity 63
5.2 Formal Specification Styles 63
5.3 Formal Specification and Life Cycle 65
5.4 The Detection of Errors in Formal Specification 66
5.5 The Utility of Formal Specification 68
5.6 A Partial SAFER Specification 71

6 Formal Analysis **79**
6.1 Automated Deduction 79
6.1.1 Background: Formal Systems and Their Models 80
6.1.1.1 Proof Theory 80
6.1.1.2 Model Theory 82

6.1.1.3 An Example of a First-Order Theory 83
6.1.2 A Brief History of Automated Proof 84
6.1.3 Techniques Underlying Automated Reasoning 87
6.1.3.1 Calculi for First-Order Predicate Logic 87
6.1.3.1.1 Normal Forms 87
6.1.3.1.2 The Sequent Calculus 88
6.1.3.1.3 The Resolution Calculus 93
6.1.3.2 Extending the Predicate Calculus 94
6.1.3.2.1 Reasoning about Equality 94
6.1.3.2.2 Reasoning about Arithmetic 96
6.1.3.2.3 Combining First-Order Theories 97
6.1.3.3 Mechanization of Proof in the Sequent Calculus 97
6.1.4 Utility of Automated Deduction 102
6.2 Finite-State Methods 103
6.2.1 Background 103
6.2.1.1 Temporal Logic 104
6.2.1.2 Linear Temporal Logic (LTL) 106
6.2.1.3 Branching Time Temporal Logic 106
6.2.1.4 Fixed Points 109
6.2.1.5 The Mu-Calculus 110
6.2.2 A Brief History of Finite-State Methods 111
6.2.3 Approaches to Finite-State Verification 113
6.2.3.1 The Symbolic Model Checking Approach 113
6.2.3.2 The Automata-Theoretic Approach 116
6.2.3.2.1 Language Containment 116
6.2.3.2.2 State Exploration 117
6.2.3.2.3 Bisimulation Equivalence and Prebisimulation Preorders 119
6.2.4 Utility of Finite-State Methods 120
6.3 Direct Execution, Simulation, and Animation 120
6.3.1 Observational Techniques 121
6.3.2 Utility of Observational Techniques 122
6.4 Integrating Automated Analysis Methods 123
6.5 Proof of Selected SAFER Property 123
6.5.1 The PVS Theory `SAFER_properties` 124
6.5.2 Informal Argument for Lemma `max_thrusters_sel` 127

7 Conclusion **131**
7.1 Factors Influencing the Use of Formal Methods 131
7.2 The Process of Formal Methods 132
7.3 Pairing Formal Methods, Strategy, and Task 133

7.4 Formal Methods and Existing Quality Control and Assurance Activities 134
7.5 Formal Methods: Verification Versus Validation and Exploration 135

References 137

A Glossary of Key Terms 167
A.1 Acronyms 167
A.2 Terms 168

B Further Reading 171
B.1 Technical Background: Mathematical Logic 171
B.2 Specification 172
B.3 Model Checking 172
B.4 Theorem Proving 173
B.5 Models of Computation 173
B.6 Applications and Overviews 173
B.7 Tutorials 174

C Extended Example: Simplified Aid for EVA Rescue (SAFER) 177
C.1 Overview of SAFER 177
  C.1.1 History, Mission Context, and System Description 177
  C.1.2 Principal Hardware Components 179
    C.1.2.1 Backpack Propulsion Module 179
    C.1.2.2 Hand Controller Module (HCM) 179
    C.1.2.3 Battery Pack 181
    C.1.2.4 Flight Support Equipment 181
  C.1.3 Avionics 181
  C.1.4 System Software 182
    C.1.4.1 Software Interfaces 182
    C.1.4.2 Maneuvering Control Subsystem 184
    C.1.4.3 Fault Detection Subsystem 185
C.2 SAFER EVA Flight Operation Requirements 188
  C.2.1 Hand Controller Module (HCM) 188
    C.2.1.1 Display and Control Unit (DCU) 188
    C.2.1.2 Hand Controller Unit (HCU) 189
  C.2.2 Propulsion Subsystem 190
  C.2.3 Avionics Assemblies 190
    C.2.3.1 Inertial Reference Unit (IRU) 190
    C.2.3.2 Power Supply Assembly (PSA) 190
    C.2.3.3 Data Recorder Assembly (DRA) 191
  C.2.4 Avionics Software 191
  C.2.5 Avionics Software Interfaces 192

- C.3 Formalization of SAFER Requirements 193
  - C.3.1 PVS Language Features 194
  - C.3.2 Overview of Formalization 195
    - C.3.2.1 Basic Types 196
    - C.3.2.2 Hand Controller Module 197
    - C.3.2.3 Propulsion Module 197
    - C.3.2.4 Automatic Attitude Hold 197
    - C.3.2.5 Thruster Selection 197
    - C.3.2.6 Avionics Model 198
  - C.3.3 Full Text of PVS Theories 198
- C.4 Analysis of SAFER 219
  - C.4.1 Formulating System Properties 219
    - C.4.1.1 Formalization of the Maximum Thruster Property . . . 220
    - C.4.1.2 PVS Theory for Maximum Thruster Property 220
  - C.4.2 Proving System Properties 222
    - C.4.2.1 Proof Sketch of the Maximum Thruster Property . . . 223
    - C.4.2.2 PVS Proof of Maximum Thruster Property 225

List of Figures

2.1 The Range of Formal Methods Options Summarized in Terms of (a) Levels of Formalization and (b) Scope of Formal Methods Use. 7
2.2 Mechanical Support for Specification and Analysis Phases of FM. 13
2.3 Front and back views of SAFER system worn by NASA crewmember. . 14

4.1 Implementation of a Full Adder. 36
4.2 Abstract State Machine Model. 36
4.3 A-7 Model of a Simple Control System. 37
4.4 State-Update and Actuator Functions within Control System. 38
4.5 Object Model of Cassini Generic Fault Protection Monitor. 42
4.6 Functional Model of Cassini Generic Fault Protection Monitor. 43
4.7 Dynamic Model of Cassini Generic Fault Protection Monitor. 44
4.8 AAH Control System State-Update and Actuator Functions. 50
4.9 Labeled AAH Pushbutton State Transition Diagram. 51

6.1 Burch *et al.*'s Mu-Calculus Model Checking Algorithm. 114
6.2 A Simple SMV Program [McM93, p. 63]. 115
6.3 Dependency Hierarchy for `SAFER_properties`. 128
6.4 Proof Tree for `SAFER_properties_max_thrusters_sel`. 129
6.5 Revised Proof Tree for `SAFER_properties_max_thrusters_sel`. 130

C.1 SAFER use by an EVA crewmember. 227
C.2 Propulsion module structure and mechanisms. 228
C.3 SAFER thrusters and axes. 229
C.4 Hand controller module. 230
C.5 Hand controller translational axes. 231
C.6 Hand controller rotational axes. 232
C.7 SAFER system software architecture. 233
C.8 AAH pushbutton state diagram. 234

List of Tables

C.1 SAFER sensor complement. 183
C.2 Thruster select logic for X, pitch, and yaw commands. 186
C.3 Thruster select logic for Y, Z, and roll commands. 187

Chapter 1

Introduction

This guidebook, the second of a two-volume series, is intended to facilitate the transfer of formal methods to the avionics and aerospace community. The first volume concentrates on administrative and planning issues [NASA-95a], and the second volume focuses on the technical issues involved in applying formal methods to avionics and aerospace software systems. Hereafter, the term "guidebook" refers exclusively to the second volume of the series. The title of this second volume, *A Practitioner's Companion*, conveys its intent. The guidebook is written primarily for the nonexpert and requires little or no prior experience with formal methods techniques and tools. However, it does attempt to distill some of the more subtle ingredients in the productive application of formal methods. To the extent that it succeeds, those conversant with formal methods will also find the guidebook useful. The discussion is illustrated through the development of a realistic example, relevant fragments of which appear in each chapter.

The guidebook focuses primarily on the use of formal methods for analysis of requirements and high-level design, the stages at which formal methods have been most productively applied. Although much of the discussion applies to low-level design and implementation, the guidebook does not discuss issues involved in the later life cycle application of formal methods. The example provided in the guidebook is based on the control function for the Simplified Aid for EVA (Extravehicular Activity) Rescue [SAFER94a, SAFER94b], hereafter referred to as SAFER[1], which has been specified and analyzed using the PVS specification language and interactive proof checker [ORSvH95]. PVS has been selected because it has been successfully used on NASA projects, including [LR93a, NASA93, LA94, Min95, BCC$^+$95, HCL95, SM95b, DR96, ML96], and because it is representative of a class of tools that offers a formal specification language in a comprehensive environment, including automated proof support. In formalizing the

---

[1] SAFER is a descendent of the Manned Maneuvering Unit (MMU) [MMU83]. The main difference between SAFER and the MMU is that SAFER is a small, lightweight, "simplified" single-string system for contingency use (self-rescue) only, whereas the MMU is a larger, bulkier, but extremely versatile EVA maneuvering device. The application of formal methods to SAFER is limited to the example in this guidebook; formal methods have not been used to support SAFER development or maintenance.

SAFER example, the priorities have been readability and portability to other formal methods paradigms. Consequently, the discussion is framed in general terms applicable to most formal methods strategies and techniques.

The guidebook is not a tutorial on formal methods; it does not provide a grounding in mathematical logic or formal specification and verification, although the appendices contain references that provide technical background, as well as a glossary of key terms. Nor is it a formal methods cookbook; there are no recipes that detail the step-by-step preparation of a formal methods product. Furthermore, the guidebook assumes that the reader is aware of the potential benefits and fallibilities of formal methods; it does not dwell on the very real benefits of the appropriate application of formal methods or the equally real pitfalls of misuse.

The guidebook does contain a fairly detailed account of the technical issues involved in applying formal methods to avionics and aerospace software systems, including a well-developed example. In order of presentation, the topics covered in the guidebook include requirements, models, formal specification, and formal analysis. However, the application of formal methods is not an essentially linear process. Formal methods are most productive when they are integrated with existing life cycle processes, and when they use an iterative strategy that successively refines and validates the formalization, the requirements, the design, and if desired, critical parts of the implementation.

This guidebook is organized as follows: Chapter 2 reviews technical considerations relevant to projects considering the use of formal methods, touching briefly on general elements of the somewhat elusive *method* underlying formal methods. This chapter also provides background material on the SAFER example developed in subsequent chapters. Chapter 3 examines the notion of requirements from a formal methods perspective and introduces selected requirements for the ongoing SAFER example. The concept of models and a survey of modeling strategies are introduced in Chapter 4, along with a formal model for a SAFER subsystem. A fragment of the specification for the SAFER requirements introduced in Chapter 3 is developed using the model defined in Chapter 4. Chapter 5 provides a discussion of formal specification, including topics ranging from specification languages, paradigms, and strategies, to type consistency of specifications. Again, a discussion of the pertinent step in the development of the SAFER example appears at the end of the chapter. Chapter 6 considers techniques and tools for formal analysis, including such topics as the role of formal proof, the impact of specification strategy on formal analysis, and the utility of various analysis strategies. A discussion of formal analysis of key properties of the SAFER specification appears at the end of the chapter. Following concluding remarks in Chapter 7 are three appendices: Appendix A contains a glossary of key terms and concepts, Appendix B lists material for further reading, and Appendix C offers an extended discussion of the complete SAFER example.

There are several ways to use this guidebook. The heart of the discussion appears in Chapters 4, 5, and 6. Readers new to formal methods may want to concentrate on these key chapters, along with the first three chapters and the conclusion, possibly skipping Chapter 6 the first time through. In most cases, historical observations and more technical material are bracketed with the “dangerous bend” signs: ☡. . . ☡ .[2] More experienced practitioners may want to focus on Chapters 5 and 6, or skip directly to the full treatment of the example in Appendix C. The SAFER example that concludes each chapter should be used to further clarify the discussion as the reader proceeds, rather than saved as a finale at the end of the chapter.

---

[2] The “dangerous bend” icon was introduced by Knuth [Knu86].

Chapter 2

The Practical Application of Formal Methods

The practical application of formal methods typically occurs within the context of a project and, possibly, within a broader context dictated by institutionalized conventions or criteria. These contexts determine the role of formal methods and the dimensions of its use. This chapter contains a review of these contextual factors, including a brief overview of the formal methods process. The discussion moves from the explicitly *formal* nature of formal methods to the more elusive *methods* implied in its use. The chapter also provides sufficient background information on SAFER to clarify and motivate pertinent aspects of the formalization and analysis of SAFER that illustrate the discussions in each of the subsequent chapters.

2.1 What Are Formal Methods?

The term *Formal Methods* refers to the use of techniques from logic and discrete mathematics in the specification, design, and construction of computer systems and software. The word "formal" derives from formal logic and means "pertaining to the structural relationship (i.e., form) between elements." *Formal logic* refers to methods of reasoning that are valid by virtue of their form and independent of their content. These methods rely on a discipline that requires the explicit enumeration of all assumptions and reasoning steps. In addition, each reasoning step must be an instance of a relatively small number of allowed rules of inference. The most rigorous formal methods apply these techniques to substantiate the reasoning used to justify the requirements, or other aspects of the design or implementation of a complex or critical system. In formal logic, as well as formal methods, the objective is the same: reduce reliance on human intuition and judgment in evaluating arguments. That is, reduce the acceptability of an argument to a *calculation* that can, in principle, be checked mechanically, thereby replacing the inherent subjectivity of the review process with a *repeatable* exercise. Less rigorous formal methods[1] tend to emphasize the formalization and forego the calculation.

This definition implies a broad spectrum of formal methods techniques, as well as a similarly wide range of formal methods strategies[2]. The interaction of the techniques and strategies yields many formal methods options, constrained, for any given project, by the role of formal methods and the resources available for its application. The roles of formal methods are discussed in the following section. An evaluation of resources as a factor shaping formal methods can be found in Volume I of this Guidebook [NASA-95a].[3] The purpose of the next few sections is to emphasize the versatility of formal methods and the importance of customizing the use of formal methods to the application.

2.2 Roles of Formal Methods

As noted above, formal methods may be used to calculate. For example, a formal method may be used to determine whether a certain description is internally consistent, whether certain properties are consequences of proposed requirements, whether one level of design implements another, or whether one design is preferable to another. In such cases, the focus of formal methods use is largely analytical. Formal methods may also have a primarily descriptive focus, for example, to clarify or document requirements or high-level design, or to facilitate communication of a requirement or design during inspections or reviews. Each use reflects a particular formal methods role. Formal methods may also be used to satisfy standards or to provide assurance or certification data, in which case the role of formal methods, as well as the analytic or descriptive content of the formal methods product, is prescribed.

The intended role or roles specified for a particular application of formal methods serves to constrain the set of techniques and strategies appropriate for that project.

2.3 Formal Methods: Degree of Formalization and Scope of Use

Formal methods options may be classified in terms of techniques that are differentiated by degree or level of formalization (Figure 2.1(a)), and strategies that are characterized by the scope of formal methods use (Figure 2.1(b)). Level of formalization and scope of use are independent factors that combine to determine the range of formal methods options, hence their juxtaposition in Figure 2.1.

---

[1] Or, equivalently, the use of a rigorous formal method at a lower level of rigor. The extent of formalization and level of rigor are discussed in Section 2.3.

[2] As used here and throughout the remainder of the guidebook, “formal methods strategies” refer to strategems for productively employing the mathematical techniques that comprise formal methods.

[3] The material in the following sections reflects the type of technical issues typically raised in a general discussion of formal methods use. More complete exploration of these and related topics can be found, for example, in [Rus93a, BS93, HB95b].

Levels of Formalization
1. Mathematical concepts and notation, informal analysis (if any), no mechanization
2. Formalized specification languages, some mechanized support
3. Formal specification languages, comprehensive environment, including automated proof checker/theorem prover

Scope of FM Use
Life cycle phases: all/selected
System components: all/selected
System functionality: full/selected

Figure 2.1: The Range of Formal Methods Options Summarized in Terms of (a) Levels of Formalization and (b) Scope of Formal Methods Use.

2.3.1 Levels of Formalization

Formal methods techniques may be defined at varying levels, reflecting the extent to which a technique formulates specifications in a language with a well-defined semantics, explicitly enumerates all assumptions, and reduces proofs to applications of well-defined rules of inference. Increasing the degree of formality allows specifications and assumptions to be less dependent on subjective reviews and consensus and more amenable to systematic analysis and replication. There is a distinction to be drawn between the terms rigor and formality; it is possible to be rigorous, that is, painstakingly serious and careful, without being truly formal in the mathematical sense. Since it is difficult to use a high degree of formality with pencil and paper [RvH93], increasing formality is associated here with increasing dependence on computer support.

As techniques mature and acquire automated support, their level of formalization typically changes. The evolution of the A-7 or Software Cost Reduction (SCR) methodology illustrates this process. In the late 1970s, Parnas, Heninger, and colleagues at the Naval Research Laboratory (NRL) defined a tabular method to specify software system requirements [$\mathrm{H}^{+}78$]. Van Schouwen subsequently formalized the methodology and its underlying mathematical model [vS90]. Researchers at NRL have continued to work on the SCR methodology, refining the model, providing a formal semantics, developing automated tools including consistency and completeness checkers, and, most recently, exploiting extant model checkers and theorem provers [HBGL95, BH97, AH97].

The levels of formalization are defined below, listed in order of increasing formality and effort. The purpose of this classification is to identify broad classes of formal methods. The distinctions underlying the classification are neither hard and fast, nor a measure of the inherent merit or mathematical sophistication of a technique. Instead, the distinctions reflect the extent to which a technique is both mathematically well-defined and supported by mechanized tools, yielding systematic analyses and replicable results.

1. The use of notations and concepts derived from logic and discrete math to develop more precise requirements statements and specifications. Analysis, if any, is informal. This level of formal methods typically augments existing processes without imposing wholesale revisions. Examples include early formulations of the A-7 methodology [H$^+$78, Hen80, vS90], various case- and object-oriented modeling techniques [Boo91, CY91b, CY91a, RBP$^+$91, Sys92], and Mills and Dyer's Cleanroom methodology [Mil93, Lin94], although the latter is an exception in that it supplants rather than augments existing processes.

2. The use of formalized specification languages with mechanized support tools ranging from syntax checkers and prettyprinters to typecheckers, interpreters, and animators. This level of formality usually includes support for modern software engineering constructs with explicit interfaces, for example, modules, abstract data types, and objects. Historically, tools at this level haven't offered mechanized theorem proving, although recent evolution of the following tools has increased their support for mechanized proof: Larch [wSJGJMW93], RAISE [Gro92], SDL [BHS91], VDM [Jon90], Z [Spi88, Wor92] and SCR [FC87, HJL95, HLK95, HBGL95].

3. The use of formal specification languages with rigorous semantics and correspondingly formal proof methods that support mechanization. Examples include HOL [GM93], Nqthm [BM88], ACL2 [KM96], EVES [CKM$^+$91], and PVS [ORSvH95]. State exploration [Hol91, ID93], model checking [McM93], and language inclusion [Kur94] techniques also exemplify this level, although these technologies use highly specialized, automatic theorem provers that are limited to checking properties of finite-state systems or of infinite-state systems with certain structural regularities.

One of the maxims of this guidebook is the importance of tailoring the use of formal methods to the task. In this case, the maxim implies that higher levels of rigor are not necessarily superior to lower levels. The highest level of formality may not be the most appropriate or productive for a given application. A project that intends using formal methods primarily to document the emerging requirements for a new system component would make very different choices than if they were formally verifying key properties of an inherently difficult algorithm for a distributed protocol. Implicit in the discussion is the importance of selecting a formal methods tool appropriate to the task. A full discussion of factors influencing tool selection can be found in [Rus93a], and a summary is available in Volume I of this guidebook [NASA-95a].

2.3.2 Scope of Formal Methods Use

The three most commonly used variations in the scope of formal methods application are listed here; others are certainly possible.

1. *Stages of the development life cycle*
   Generally, the biggest payoff from formal methods use occurs in the early life cycle stages, given that errors cost more to correct as they proceed undetected through the development stages; early detection leads to lower life cycle costs. Moreover, formal methods use in the early stages provides precision precisely where it is lacking in conventional development methods.

2. *System components*
   Criticality assessments, assurance considerations, and architectural characteristics are among the key factors used to determine which subsystems or components to analyze with formal methods. Since large systems are typically composed of components with widely differing criticalities, the extent of formal methods use should be dictated by project-specific criteria. For example, a system architecture that provides fault containment for a critical component through physical or logical partitioning provides an obvious focus for formal methods activity and enhances its ability to assure key system properties.

3. *System functionality*
   Although formal methods have traditionally been associated with "proof of correctness," that is, ensuring that a system component meets its functional specification, they can also be applied to only the most important system properties. Moreover, in some cases it is more important to ensure that a component does *not* exhibit certain negative properties or failures, rather than to prove that it has certain positive properties, including full functionality.

2.4 Reasonable Expectations for Formal Methods

A formal method is neither a panacea, nor a guarantee of a superior product. Realistic expectations are a function of the designated role(s) and extent of formal methods use and of the project resources allocated to the formal methods activity. Judicious, skillful application of formal methods can detect faults *earlier* than standard development processes, thereby greatly reducing the incidence of mistakes in interpreting, formalizing, and implementing correct requirements and high-level designs. Because formal methods encourage a systematic enumeration and exploration of cases, they encourage the early discovery of faults in requirements or high-level designs that would otherwise be discovered only during programming. Of course, the same claim can be made for pseudocode, dataflow diagrams, or other quasi-formal notations that can be used early in the life cycle.

The advantage of formal methods is that by concentrating on what is required, they focus more directly on the topic of interest and avoid the distractions entailed by implementation factors. Stronger claims can even be made for fully formal techniques. Equally judicious, skillful applications of *the most rigorous formal methods* can detect more faults than would otherwise be the case and, in certain circumstances, subject to certain caveats, they can also *guarantee* the absence of certain faults. In particular, by working early in the life cycle, on reasonably abstracted representations of the hardest part(s) of the overall problem, the highest-level formal methods can validate crucial elements of the requirements or high-level design. Finally, in contrast to such techniques as direct execution, prototyping, and simulation, which can explore a large, but necessarily incomplete set of system behaviors, deductive formal methods and state exploration techniques support exhaustive examination of *all* behaviors.[4] The extent to which a project realizes some or all of the benefits described here depends on the availability of essential resources, the skill with which formal methods use is tailored to the application, and the degree to which the expectations fit the dimensions of the project.

2.5 The Method Underlying Formal Methods

> *In the context of an engineering discipline, a* ***method*** *describes the way in which a process is to be conducted. In the context of system engineering, a method is defined to consist of (1) an underlying model of development, (2) a language, or languages, (3) defined, ordered steps, and (4) guidance for applying these in a coherent manner.*
>
> *Most so-called formal methods do not address all of these issues.... Indeed, the formal methods community has been slow to address such methodological aspects.*[5] [HB95b, p. 2]

Although the four elements in the preceding definition may be somewhat controversial, the observation that there is a paucity of *method* in formal methods is not. The observation focuses in particular on the apparent absence of "defined, ordered steps" and "guidance" in applying those methodical elements that have been identified. One reason for the absence of method is that the intellectual discipline involved in modeling, specification, and verification eludes simple characterization; the intuition that guides effective abstraction, succinct specification, and adroit proof derives from skill, talent, and experience and is difficult to articulate as a process.

Exceptions to this observation include specialized methodologies for particular application areas, such as the area of embedded systems — reactive systems that operate continuously and interact with their environment, including Parnas's "four variable method" [vS90, vSPM93], NRL's Software Cost Reduction (SCR) method [FC87, HBGL95], the Software Productivity Consortium's Requirements Engineering (CoRE) method [FBWK92], and Harel's Statecharts [Har87, H$^+$90] and its derivatives, such as Leveson's Requirements State Machine Language (RSML) [LHHR94]. Historically, the methods developed for reactive systems have provided organizing principles, conceptual models, and in many cases, specification languages, and systematic checks for well-formedness of specifications. Although many of these methodologies provide some mechanized analysis and are currently exploring additional mechanized checks, few have yet to provide the range of analysis available in a true theorem prover or proof checker.

---

[4] State exploration techniques require a "downscaled" or finite state version of the system and typically involve a more concrete representation than that used with theorem provers or proof checkers. These and related topics are discussed in Chapter 6.

[5] The material quoted here is based on a discussion in [Kro93].

Although the method implied in formal methods has been slow to emerge (with the exception of the methodologies noted above), broad outlines that effectively constitute an "underlying model of development" are worth noting. The process of applying formal methods to a chosen application typically involves the following phases: *characterizing the application*, *modeling*[6], *specification*, *analysis (validation)*, and *documentation*. The distinction between phases is somewhat artificial and should not be taken too literally. For example, it is difficult and not particularly instructive to determine precisely where modeling ends and specification begins. Each phase consists of constituent processes. Again, the enumeration below is suggested, not prescribed, and the overall process (i.e., the four constituent phases) is iterative rather than sequential. For example, characterization of the application may be influenced by consideration of potential models, the process of specifying the application may suggest changes to the underlying model, or the process of verifying a key property may trigger changes to the specification or even to the underlying model. Ideally, documentation accompanies all the phases summarized here:

- The Characterization Phase: Synthesize a thorough understanding of the application and the application domain.
  - Conduct a thorough study of the application, noting key components and subcomponents, interfaces, essential algorithms, fundamental behaviors (nominal and off-nominal), data and control flows, and operational environment.
  - Identify and study related work, if any.
  - Acquire additional knowledge of the application domain, as needed.
  - Integrate the accumulated knowledge into a working characterization of the application. Some practitioners, especially those working alone, tend to "internalize" an application, working strictly from mental notes. Other practitioners produce working documents and notes. The culture in which a project operates in large part determines the artifacts (if any) of this phase. Still, the importance of this phase should not be underestimated; total immersion in an application is crucial for developing insight into the most appropriate models and the most appropriate specification and validation strategies. In some cases, such as hardware verification, there is considerable precedent and there are fairly well-established paradigms. There is also a standard

---

[6] As used here, the term "model" refers to the mathematical representation of a system that underlies the system's specification. In this usage, the "models" checked by state exploration tools or model checkers are viewed as specifications.

paradigm for proving hierarchical specification chains, that is, hierarchies of specifications at different levels of abstraction (see Section 5.3). However, in most other cases, there is often little applicable precedent and there are few, if any, established paradigms.

- The Modeling Phase: Define a mathematical representation suitable for formalizing the application domain and for calculating and predicting the behavior of the application in that context. (See Chapter 4.)
  - Evaluate potential mathematical representations, considering such general factors as the level of abstraction, generality, expressiveness, analytical power, and simplicity, as well as specific factors, such as the computational model, and explicit (implicit) representation of state and time. Mechanized tool support, if any, may also be a factor. The logic underlying a tool may support the use of certain mathematical representations and discourage the use of others.
  - Select the mathematical representation most suitable for the application.
  - Model key elements of the application and their relationships. As noted above, this (sub)process transitions into the specification phase.
- The Specification Phase: Formalize relevant aspects of the application and its operational environment. (See Chapter 5.)
  - Develop a specification strategy, considering such factors as hierarchical (multilevel) versus single-level specification, constructive versus descriptive specification style (see Section 5.2), and procedural and organizational issues, such as developing reusable theories and common definitions, and specification chronology.
  - Using the chosen model and specification strategy, compose the specification.
  - Analyze the syntactic and semantic correctness of the specification.
- The Analysis Phase: Validate the specification. (See Chapter 6.)
  - Interpret or execute the specification.
  - Prove key properties and invariants.
  - Establish the consistency of axioms, if any.
  - Establish the correctness of hierarchical layers, if any.
- The Documentation Phase: Record operative assumptions, motivate critical decisions, document the rationale and crucial insights, provide explanatory material, trace specification to requirements (high-level design), track level of effort, and where relevant, collect cost/benefit data.

- Maintenance and Generalization: Revisit and modify the specification and its analysis as required, for example, to predict the consequences of proposed changes to the modeled system, to accommodate mandated changes to the modeled system, to support reuse of the formal specification and analysis, or to distill general principles from the formalization and analysis.

Formal methods are supported in the specification and analysis phases with mechanized tools that perform the steps shown in Figure 2.2. Tools that support user interaction typically provide these steps explicitly, whereas tools that are fully automated do so implicitly. For example, most state exploration tools are fully automatic and do not provide user control of the steps that check for syntactic and semantic consistency. Mechanized support for the modeling phase exists, for example, in some of the informal object-oriented methodologies and in methods such as SCR. However, mechanized support for modeling is not (yet) included in most formal methods (FM) systems and is therefore not represented in Figure 2.2.

FM Phase	*Tool*	*Tool Function*
Specification	Parser	Checks syntactic consistency
Specification	Unparser	Translates internal representation into display and outputs formatted text
Specification	Typechecker	Checks semantic consistency
Analysis	Animator, simulator	Exhibits behavior of system modeled by syntactically and semantically correct specification
Analysis	Proof checker, model checker	Performs proof over syntactically and semantically correct specification

Figure 2.2: Mechanical Support for Specification and Analysis Phases of FM.

Except for documentation and maintenance, all the phases listed above form the core of subsequent chapters, beginning with the characterization phase. This chapter concludes with background regarding SAFER drawn from requirements documents and operations manuals typical of the kind of documentation used for developing an initial characterization of an application and its domain.

2.6 An Introduction to SAFER

Unless otherwise noted, this section is based on the SAFER Operations Manual [SAFER94a]. A more detailed version of the material, along with all figures cited in this discussion, can be found in Appendix C.

SAFER, as shown in Figure 2.3, is a small, lightweight propulsive backpack system designed to provide self-rescue capability to a NASA space crewmember separated

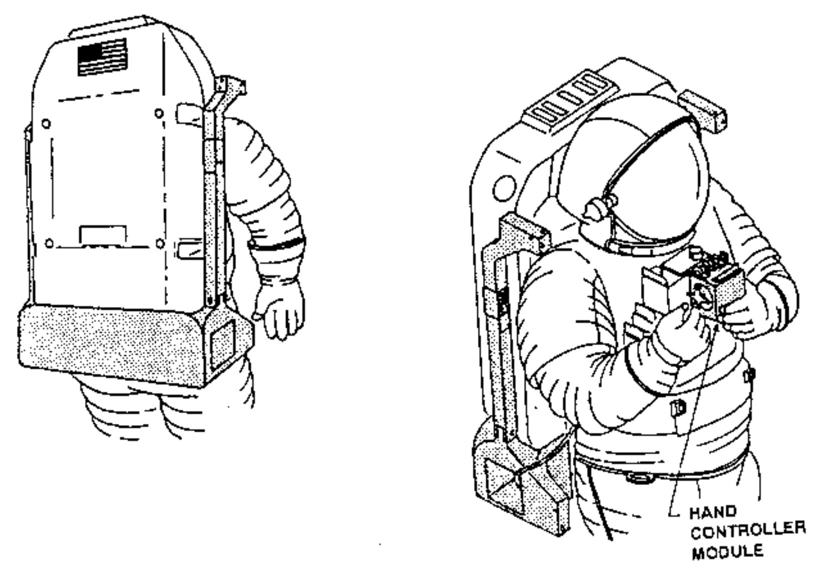


Figure 2.3: Front and back views of SAFER system worn by NASA crewmember.

during an EVA. This could be necessary if a safety tether broke or was not correctly fastened during an EVA on a space station or on a Space Shuttle Orbiter docked to a space station. SAFER provides an attitude hold capability and sufficient propellant to automatically detumble and (manually) return a separated crewmember. A flight test version of SAFER was flown on STS-64 and STS-76, and production variants have been used on the initial MIR docking flights.

The SAFER flight unit weighs approximately 85 pounds and folds for launch, landing, and on-orbit stowage inside the Orbiter airlock. SAFER attaches to the underside of the Extravehicular Mobility Unit (EMU) primary life-support subsystem backpack, without limiting suit mobility and is controlled by a single hand controller attached to the EMU display and control module.

The hand controller contains a small liquid crystal display (LCD), two light-emitting diodes (LEDs), a small control unit with three toggle switches, and the hand controller grip, as shown in Figure C.4. The displays and switches are visible from all possible head positions inside the EMU helmet, and the switches are positioned for either left- or right-handed operation. The functions of the three displays and three switches are as follows:

1. Liquid Crystal Display: A 16-character, backlit LCD displays prompts, status information, and fault messages.

2. Light-emitting Diode: A red LED labeled "THR" lights whenever a thruster-on condition is detected by the control software.

3. Light-emitting Diode: A green LED labeled "AAH" lights whenever automatic attitude hold is enabled for one or more rotational axes.
4. Switch: A three-position toggle switch labeled "PWR" powers on SAFER and initiates the self-test or activation test functions.
5. Switch: A three-position momentary toggle switch labeled "DISP" controls the LCD display, allowing the crewmember to select the previous or next parameter, message, or test step. The switch springs back to the center (null) position when released.
6. Switch: A two-position toggle switch labeled "MODE" selects the hand controller mode associated with rotation and translation commands.

The hand controller is a four-axis mechanism with three rotary axes and one transverse axis. To generate a command, the crewmember moves the hand controller grip (mounted on the right side of the hand controller module) from the null center position to mechanical hardstops on the hand controller axes. To terminate a command, the crewmember returns the hand controller to the center position or releases the grip so that it automatically springs back to the center. Figures C.5 and C.6 illustrate the hand controller axes for translational and rotational commands, respectively. For example, Figure C.5 indicates that with the control switch set to translation mode, $\pm Y$ commands are generated by pulling or pushing the grip right or left, respectively. Careful study of these figures reveals that the X translation command and the pitch rotation command are always available in either mode. A pushbutton switch on the top of the hand controller grip initiates and terminates automatic attitude hold.

The avionics software processes inputs from the hand controllers and various sensors, and includes the following components:

1. Control Electronics Assembly (CEA): The CEA microprocessor takes inputs from sensors and hand controller switches and actuates the appropriate thruster valves.
2. Inertial Reference Unit (IRU): The IRU senses angular rates and linear accelerations and is central to the attitude hold capability.
3. Data Recorder Assembly (DRA): The DRA collects flight-performance data, hand controller and automatic attitude-hold commands, and thruster firings.
4. Valve Drive Assemblies (VDAs): Each of the four VDAs, located with a cluster of six thrusters, takes firing commands from the CEA and applies voltages to the selected valves.
5. Power Supply Assembly (PSA): The PSA produces regulated electrical power for all SAFER electrical components.
6. Instrumentation Electronics: SAFER instrumentation includes a variety of sensors, all of which are listed in Table C.1.

The avionics software has two principal functions: maneuvering control for both commanded accelerations and automatic attitude hold actions, and fault detection, which supports inflight operation, pre-EVA checkout, and ground checkout. A brief summary of the control function is presented here. Sections C.1.4.2 and C.1.4.3 present a more detailed summary of the maneuvering control function and an account of the fault detection function, respectively.

The maneuvering-control software commands both rotational and translational accelerations. Translation commands provide acceleration along a single translational axis and are prioritized so that X is first, Y is second, and Z is third. When rotation and translation commands are present simultaneously, rotation takes priority and translations are suppressed. Conflicting input commands result in no output to the thrusters. Whenever possible, acceleration is provided as long as a hand controller or automatic attitude-hold command is present.

The SAFER crewmember can initiate (single-click) or terminate (double-click) automatic attitude hold at any time via the pushbutton on the top of the hand controller grip. When terminated, automatic attitude hold is disabled for all three rotational axes. If a crewmember issues a rotational command for a given axis when automatic attitude hold is active, it is immediately disabled for that axis only. However, to ensure that a failed-on hand controller command in a rotational axis will not disable automatic attitude hold on that axis, automatic attitude hold takes precedence over a crewmember-issued rotational command if the two are initiated simultaneously. Automatic attitude hold provides an automatic rotational deceleration until all three axis rates are near zero. These near-zero rates are automatically maintained whenever automatic attitude hold is active.

Thruster-select logic takes acceleration commands from the hand controller and from the automatic attitude-hold function, creates a single acceleration command, and chooses thruster firings to achieve the commanded acceleration. Thruster selection results in on-off commands for each thruster, with a maximum of four thrusters turned on simultaneously. Thruster arrangement and designations are shown in Figure C.3. Tables C.2 and C.3 specify the selection logic.

SAFER has 24 gaseous nitrogen (GN_2) thrusters — four thrusters pointing in each of the $\pm X$, $\pm Y$, and $\pm Z$ axes. The thrusters are arranged in four groups of six thrusters each, located as shown in Figure C.3. As noted, thruster valves open, causing the thrusters to fire in response to directives from the avionics subsystem, which commands as many as four thrusters at once to provide six degree-of-freedom maneuvering control ($\pm X$, $\pm Y$, $\pm Z$, $\pm roll$, $\pm pitch$, $\pm yaw$). The SAFER propulsion system provides a total delta velocity of at least 10 feet per second with an initial charge. The four GN_2 tanks have a relatively small capacity and require several recharges during an EVA. The recharge station is located in the Orbiter payload bay. When SAFER is not in use or if a malfunction (such as a failed-on thruster) occurs, the tanks can be isolated via a manually actuated isolation valve.

The SAFER example introduced here is used throughout the guidebook to illustrate key points in each chapter. Although this example attempts to formalize the actual SAFER design, pragmatic and pedagogical considerations have inevitably resulted in differences between the actual design and the formal specification. These differences do not detract from the presentation of a realistic example that captures the basic characteristics of a class of space vehicles and the computerized systems that control them. The fragment of the example chosen for inclusion at the end of each subsequent chapter focuses on the thruster selection function responsible for creating an integrated acceleration command from hand controller and automatic attitude-hold inputs.

Chapter 3

Requirements

Requirements define the set of conditions or capabilities that must be met by a system or system component to satisfy a contract, standard, or other formally imposed document or description [SE87]. For example, IEEE Standard 1498 [IEEE194, p. 7] defines a requirement as "a characteristic that a system or software item must possess in order to be acceptable to the acquirer." Similarly, the NASA *Guidebook for Safety Critical Software Analysis and Development* [NASA-96, p. A-18] defines software requirements as "statements describing essential, necessary, or desired attributes." In the context of this guidebook, requirements are taken to be a statement of the essence of a system that is typically produced at or near the beginning of the life cycle and guides and informs the development, implementation, and maintenance of that system.[1] The number of steps between requirements, capture, and implementation depends on the life cycle process for the system. Arguably, the more clearly articulated and differentiated the life cycle phases are, the more likely it is that the requirements statement will be suitable for formal analysis. A well-defined life cycle reflects a mature process, including an appreciation for the role and task of quality assurance. For example, a fairly typical, mature life cycle process might include requirements definition, system design, high-level design, low-level design, coding, testing (unit testing, component or function testing, system testing), user support, and maintenance.

There are many considerations in the elicitation, capture, modeling, specification, validation, maintenance, traceability, and reuse of requirements, and a burgeoning group of researchers interested in addressing these and related issues. This activity has led to the recent emergence of a "discipline" [FF93, p. vi] known as "Requirements Engineering" that attempts to establish "real-world goals for, functions of, and constraints on software systems" [Zav95, p. 214] and includes researchers in the social sciences as well as in several areas of computer science.[2]

---

[1] This and similar remarks in Section 3.1.1 are not meant to suggest a particular life cycle model.

[2] Representative papers may be found in the proceedings of several new conferences, including the biennial international symposium first held in 1993 [RE93,RE95] and the biennial international conference first held in 1994 [ICRE94,ICRE96].

3.1 Requirements and Formal Methods

This guidebook takes a less generic interest in requirements, focusing here on requirements as objects of formal analysis and, in particular, the characteristics of requirements that influence the application of formal methods, and conversely.

3.1.1 Impact of Requirements Specification on Formal Methods

The most important characteristics of requirements as objects of formal analysis are the level at which the requirements are stated, the degree to which they are explicitly and unambiguously enumerated, the extent to which they can be traced to specific system components, and the availability of additional information or expertise to provide the rationale to motivate and clarify the requirements definition (as necessary).

3.1.1.1 Level of Requirements Capture

Requirements for the early stages of the life cycle, that is, up to and including the high-level design phase, should be reasonably abstract and focus on basic characteristics, including essential behaviors and key properties of the system. At this level, implementation considerations and low-level detail tend to distract one from the basic system functionality. Requirements written at too low a level or with too strong an implementation bias may require reverse engineering before formal methods can be productively applied.

3.1.1.2 Explicitness of Requirements Statement

Requirements should also be completely, precisely, and unambiguously stated. At this level, the idea is to have a clear, precise statement that is reasonably complete and doesn't admit multiple interpretations. This appears to contradict the previous point, that the requirements be reasonably abstract and distill only essential behaviors and properties, but there is really no contradiction. Clarity, precision, and completeness involve explicitly identifying underlying assumptions and thoroughly enumerating all relevant cases rather than specifying low-level detail and implementation factors. Ambiguous requirements that cannot be further clarified may require the formal methods practitioner to define and explicitly record a set of operative assumptions to initiate the formal specification and analysis. Ultimately, any operative assumptions, as well as the requirements specification, should be validated.

3.1.1.3 Clarity of Delineation between a System and Its Environment

Requirements should clearly state the assumptions a system makes about its operating environment and should clearly delineate the boundary between the system and its operative context. For example, requirements should explicitly identify *environmental quantities* that the system measures, controls, or assumes, such as temperatures, pressures, and user interface assumptions [HB95a, p. 23].[3]

3.1.1.4 Traceability of Requirements

System-level requirements should be traceable to identifiable (functional) subsystems, components, or interfaces. Requirements that cannot be so traced may prove difficult to validate insofar as they specify system-level properties or behavior that is too general or too ill-defined to be formally analyzed.

3.1.1.5 Availability of Underlying Rationale and Intuition

Requirements should also contain background material that motivates and illuminates the requirements statement. Although such material is typically excluded from requirements documents, it is often possible to find domain expertise, project personnel, and artifacts that provide essential information and insight. Such supplemental material is crucially important if the requirements statement is low-level, implementation-oriented, incomplete, or ambiguous.

It is unusual to be handed a set of requirements that is well-suited to formal specification and analysis. Although formal methods provide techniques and tools for distilling a set of requirements from informal or quasi-formal specifications and for exposing missing or incomplete requirements, formal methods are not a panacea. The practitioner should factor in the availability and suitability of requirements documents when considering a formal methods application.

To illustrate, consider briefly the experience recounted in [NASA93], which describes an attempt to formalize the official Level C requirements for the Space Shuttle Jet-Select function [Roc91]. Although Space Shuttle flight software is exemplary among NASA software development projects, the requirements analysis and quality assurance in early life cycle phases of the Shuttle used then-current (late 1970s and early 1980s) products and tools. Shuttle software requirements are typically written as Functional Subsystem Software Requirements (FSSRs) – low-level software requirements specifications written in English prose and accompanied by secondary material including pseudocode, and diagrams and flowcharts with in-house notations. Interpreting the Jet-Select FSSR documents required the combined efforts of a multicenter team for several months and relied extensively on resident expertise at IBM Federal Systems Division.[4] When a new set of high-level Jet-Select requirements was formalized in the PVS specification language, it became clear that the Jet-Select function could be stated more simply. To validate the PVS specification, approximately a dozen lemmas, derived from a list of high-level Jet-Select properties identified by IBM, were formalized and proven. The fact that the algorithm and its essential properties are difficult to discern from the FSSRs illustrates two complementary points: (1) the potential problems of low-level requirements that only implicitly capture key properties and essential functionality, and (2) the value of supplemental sources and materials to provide crucial information, for example, the list of desired Jet-Select properties and the clarifications provided by IBM domain experts.[5]

---

[3] This paraphrase of a statement by Parnas, who has been among the most vocal advocates for an explicit delineation between a system and its environment, was made in the context of computer software systems, but the remark applies equally to other types of systems.

[4] The multicenter team consisted of personnel from NASA's Jet Propulsion Laboratory, Langley Research Center (LaRC), and Johnson Space Center, and included subcontractors from Lockheed Martin Space Mission Systems (formerly Loral, and, prior to that IBM, Houston) and SRI International. (The work cited here was completed prior to either the Loral or Lockheed Martin eras, hence the references to IBM.)

3.1.2 Impact of Formal Methods on Requirements

The application of formal methods typically produces tangible artifacts, including formal models, specifications, and analyses, that can impact the requirements to which they are applied. The nature of the impact depends on the strategy used in the requirements development process, and in particular, the degree to which formal methods are integrated into the existing process.

Fraser and his colleagues [FKV94] attempt to classify integration strategies with respect to the following factors:

1. Does the strategy go directly from the informal requirements to the formalized specification or does it introduce intermediate and increasingly formal models of the requirements?
2. If the strategy introduces intermediate (semiformal) models, is the process one of parallel, successive refinement of the requirements and the formal specification, or are the formal specifications derived after the (semiformal) requirements models have been finalized in a sequential strategy?
3. To what extent does the strategy offer mechanized support for requirements capture and formalization?

The question of mechanized support for requirements capture and formalization remains somewhat academic, since the fully automatic characterization of requirements still relies primarily on research tools with limited scope and scalability. One example is a knowledge-based "specification-derivation system" that uses difference-based reasoning and analogy mapping to recognize and instantiate schemas and interactively derive specifications in a language similar to the *Larch Shared Language* [FKV94, p. 82].

---

[5] This example also illustrates the fundamental cost/benefit trade-offs that invariably arise when substantial reverse engineering is required before formal methods can be applied. These and related planning issues are discussed in Volume I of this guidebook [NASA-95a].

Another example is the use of data-flow diagrams and decision tables to develop "Structured Analysis" specifications that are then translated in VDM specifications by means of "interactive rule-based algorithmic methods" [FKV94, pp. 84-5].[6]

Of more immediate interest are the strategies that use an iterative approach to the successive refinement of requirements. An example of the sequential application of the iterative strategy is the use of formal methods in certain re-engineering projects where the requirements are mature and well-established. However, it is the parallel application of the iterative strategy that most substantively impacts the requirements definition. An example of this type of application includes formalization of immature requirements or formalization of requirements for ill-defined or ill-structured problem domains. In these cases, there is the "potential of letting semiformal and formal specifications aid each other in a synergistic fashion during the requirements discovery and refinement process" [FKV94, p. 82]. If this synergy is positive, the formal models, specifications, and analyses may ultimately become (part of) the requirements—a development some would applaud and others would view with concern. For example, Parnas [HB95a, p. 21] notes that "Engineers make a useful distinction between specifications, descriptions, and models of products. This distinction seems to be forgotten in the computer science literature." This may be similarly applicable to requirements, models, and specifications. On the other hand, active research into formal semantics and automated reasoning frameworks for industrially used notations [BS93, p. 191] points toward a coalescence in some environments of informal requirements with their formalization and analysis.

3.2 Conventional Approaches to Requirements Validation

It is well recognized that identifying and correcting problems in the requirements and early-design phase avoids far more costly fixes later. It is often said that late life cycle fixes are 100 times more expensive than corrections during the early phases of software development [Boe87, p. 84]. Focused arguments for the utility of software-requirements analysis and validation have become increasingly common. For example, Kelly [KSH92] documents a significantly higher density of defects found during requirements versus later life cycle inspections. Lutz [Lut93] notes that of roughly 195 "safety-critical" faults detected during integration and system testing of the Voyager and Galileo spacecraft, 3 were programming bugs, 96 were attributed to flawed requirements, 48 resulted from incorrect implementation of the requirements, and the remaining 48 faults were traced to misunderstood interfaces.

Standard approaches to requirements analysis and validation typically involve manual processes such as "walk-throughs" or Fagan-style inspections [Fag76, Fag86]. The term *walk-through* refers to a range of activities that can vary from cursory peer reviews to formal inspections, although walk-throughs usually do not involve the replicable process and methodical data collection that characterize Fagan-style inspections. Fagan's highly structured inspection process was originally developed for hardware logic, next applied to software logic design and code, and ultimately successfully applied to artifacts of virtually all life cycle phases, including requirements development and high-level design [Fag86, p. 748]. A Fagan inspection involves a review team with the following roles: a *Moderator*, an *Author*, a *Reader*, and a *Tester*. The Reader presents the design or code to the others, systematically walking through every piece of logic and every branch at least once. The Author represents the viewpoint of the designer or coder, and the perspective of the tester is represented, as expected, by the Tester. The Moderator is trained to facilitate intensive, but constructive and optimally effective, discussion. When the functionality of the system is well-understood, the focus shifts to a search for faults, possibly using a checklist of likely errors to guide the process. The inspection process includes equally intense and highly structured rework and follow-up activities. One of the main advantages of Fagan-style inspections over other conventional forms of verification and validation is that they can be applied early in the life cycle, for example, to requirements and high-level design. Thus potential anomalies can be detected before they become entrenched in the low-level design and implementation.

---

[6] The relative immaturity of these particular activities does not reflect on the acknowledged maturity of formal methods techniques in general. See, for example, [Gla95, McI95].

NASA supports a process derived from Fagan inspections, called “Software Formal Inspections” [NASA-93b, NASA-93a] that uses teams drawn from peers involved in development, test, user groups, and quality assurance. The seven-step NASA process spelled out in [NASA-93b] consists of planning, overview, preparation, inspection meeting, third hour, rework, and follow-up stages. NASA inspections use checklists, as well as standardized forms to record product errors and collect metrics associated with the inspection process. The collection and monitoring of metrics is an integral part of NASA’s inspection process because it documents the progress of a project. If reinspection is required, several of the steps may be repeated. With small variations, the NASA inspection process is used at several NASA centers, including the Goddard Space Flight Center (GSFC), Jet Propulsion Laboratory (JPL) [Bus90], Johnson Space Center (JSC)[7], Langley Research Center (LaRC), and Lewis Research Center (LeRC). The current validation process for NASA’s Space Shuttle flight software includes close adherence to the inspection process for requirements, high-level test plans, and source code [NASA93, p. 21].

Although these processes are considered effective and the quality of NASA shuttle flight software is among the highest in NASA software development projects, the requirements analysis seems less reliable than the analyses performed on later life cycle products. For example, [Rus93a, p. 38] notes that “a quick count of faults detected and eliminated during development of the space shuttle on-board software indicates that about 6 times as many faults ‘leak’ through requirements analysis, than leak through the processes of code development and review.” In light of these and similar observations, the following characteristics of the requirements analysis process have been noted [NASA93, p. 9, 22]:

---

[7] The formal inspections cited here are actually used by Lockheed Martin Space Information Systems (formerly, Loral and, prior to that, IBM, Houston), the Space Shuttle software subcontractor.

- Current techniques are largely manual and highly dependent on the skill and diligence of individual inspectors and review teams.
- There is no methodology to guide the analysis process and no structured way for Requirement Analysts (RAs) to document their analysis. There are no completion criteria.
- Although these techniques catch a substantial number of defects, the density of defects found suggests that some errors escape detection.
- NASA projects using currently available techniques have reached a quality ceiling on critical software subsystems, suggesting that innovations are needed to reach new quality goals.

These types of issues constitute a significant part of the rationale for exploring the use of formal methods to complement and enhance existing requirements analysis and design analysis processes for critical aerospace and avionics software systems.

3.3 SAFER Requirements

The set of SAFER flight operations requirements used in this document are derived from three official project documents:

- Project Requirements Document [SAFER92]
- Prime Item Development Specification [SAFER94b]
- Operations Manual [SAFER94a]

The derivation of these requirements illustrates challenges that typically confront efforts to formalize requirements for real-world systems. For example, the Project Requirements Document provided brief characterizations for major components and functions. Requirements at this level, such as those reproduced below, provide background information, but they are at too high a level to be useful in the development of formal specifications.

- The SAFER Flight Test Article shall provide six degree-of-freedom manual maneuvering control.
- The SAFER Flight Test Article shall provide crewmember-selectable, three degree-of-freedom Automatic Attitude Hold (AAH).

The Prime Item Development Specification, while more informative, lacks detail in certain critical areas. In general, the Operations Manual, which was not intended as a requirements document, provides the most consistently useful information. Ultimately, synthesizing the material from two of the three sources was necessary first in order to characterize a system that could be meaningfully formalized. A subset of the requirements from the Prime Item Development Specification was augmented with more details from the Operations Manual. This inherently subjective process, described here, was guided by the need for requirements that provided a workable level of detail based on a well-defined system architecture. If existing requirements documents directly support the application of formal methods, or if domain expertise is readily available, the process described here would not be necessary for formalization and analysis.

The subset of the requirements presented here (numbers 37 - 42) focuses on the thruster-select function of the avionics software. Only the requirements that directly specify thruster selection have been included; those indirectly involved, such as the requirements that specify components providing thruster-selection input (the hand controller unit) and output (the propulsion subsystem), appear in Section C.2, which contains the full set of SAFER requirements.

Requirements 37 - 42 below specify the two basic thruster-select functions: (1) integrating the input from the hand controller and automatic attitude hold (AAH) into a single acceleration command and (2) selecting the set of thrusters to accomplish the command. This functionality is specified through a combination of high-level "shall" statements and lower-level tables that define the thruster-select logic. The numbers associated with each requirement correspond to those used in Appendix C.

37. The avionics software shall disable AAH on an axis if a crewmember rotation command is issued for that axis while AAH is active.

38. Any hand controller rotation command present at the time AAH is initiated shall subsequently be ignored until a return to the off condition is detected for that axis or until AAH is disabled.

39. Hand controller rotation commands shall suppress any translation commands that are present, but AAH-generated rotation commands may coexist with translations.

40. At most one translation command shall be acted upon, with the axis chosen in priority order X, Y, Z.

41. The avionics software shall provide accelerations with a maximum of four simultaneous thruster firing commands.

42. The avionics software shall select thrusters in response to integrated AAH and crew-generated commands according to Tables C.2 and C.3.

Chapter 4

Models

The term *model* is used in two different, albeit related, ways in the context of formal methods. On the one hand, "model" is used to refer to a mathematical representation of a natural or man-made system. This is consistent with the usage in science and engineering, where mathematical representations are used to predict or calculate properties of the systems being modeled. The statistical models used to analyze and predict meteorological phenomena and the models of planetary motion used to calculate satellite launch trajectories and orbits are examples of these types of mathematical models, as are the state machine models used to explore the behavior of complex hardware and software systems.

A second usage of the term "model" derives from precise terminology in formal logic and refers to a mathematical representation that satisfies a set of axioms. Exhibiting a model for a set of axioms demonstrates that the axioms are consistent. For example, one way to show that a specification is consistent is to show that its axioms have a model, as discussed in Chapter 6.

This chapter surveys characteristics of the types of mathematical models used in formal methods and concludes with a discussion on modeling the SAFER thruster selection function.

4.1 Mathematical Models

While there is no ambiguity about the meaning of the term "model" in the formal logic sense, and little confusion about its informal use in the real world of concrete objects, there is residual confusion surrounding the informal use of the term to refer to mathematical objects. For example, when speaking of real products, such as jet planes, there is no problem in distinguishing the notions of model, prototype, specification, and description. A model of a 747 may or may not be flightworthy and fit on a desk.[1] A prototype, on the other hand, would be one of the first 747s built and would exhibit most, if not all, key properties of the actual 747 aircraft, including the ability to accommodate 350 passengers. A specification of the 747 would capture certain important properties of the 747, possibly including the property that dimensions of the wing stand in a certain relationship to the overall dimensions of the plane. A description is the least constrained representation and may even include such useless detail as the fact that the plane has a rather bulbous profile.[2] On the other hand, Parnas' definition of a model as "a product, neither a description nor a specification." [Par95, p. 22] explicitly acknowledges a confusion in the context of formal methods, where models and specifications are frequently conflated. Concurrency provides a case in point. "It's not that one usually wants to specify concurrency, but rather to study the properties of a model of concurrency resulting from a specification of a system." [CS89, p. 89]

---

[1] Jackson [Jac95, pp. 120-122] follows Ackoff [Ack62] in distinguishing three kinds of model: iconic, analogic, and analytic. Using this three-way distinction, the model of the 747 is iconic, that is, the

4.1.1 Characteristics of Mathematical Models

In the context of formal methods, the most useful models tend to be abstract representations that focus on essential characteristics expressed in reasonably general terms and formalized in judiciously chosen mathematics, that is, in mathematical representations that are suitably expressive and provide sufficient analytic power. Of course, accuracy with respect to the system being modeled is also essential.

4.1.1.1 Abstraction

Exploring the relationship between modeling and specifying a concrete (physical) object, such as the 747, yields insight into desirable characteristics of abstract (mathematical) models. For example, while it is possible to build a full-scale model of the 747, it is almost certainly more useful to *abstract* away less important or less relevant features of the 747 and concentrate on the simplest or most general expression of essential features of interest. Two highly desirable consequences of creating suitably abstract models are the elimination of distracting detail and the avoidance of premature implementation commitments. For example, imagine using a desk-size model to discuss properties of the overall design, that is, the layout and proportions of the aircraft, and of certain components, such as the shape of the fore and aft sections of the wing, while ignoring properties relating to the aircraft's size or to the structural materials used to build it.

The choices of mathematical representation and level of abstraction carry inherent implications that must be explicitly considered. For example, Hayes describes the implications of certain choices for modeling a simple symbol table.

> "We are describing a symbol table by modeling it as a partial function.... Here ...we use it [the function] to describe a data structure. There may be many possible models that we can use to describe the same object. Other

---

747 model is an *icon* of a real plane. See Section 4.1.1.3 for a brief discussion of analogic and analytic models.

[2] The 747 example is based on a discussion in [Par95].

> models of a symbol table could be a list of pairs of symbol and value, or a binary tree containing a symbol and value in each node. But these other models are not as abstract, because many different lists (or trees) can represent the same function. And we would like two symbol tables to be equal if they give the same values for the same symbols." [Hay87, p. 39]

4.1.1.2 Focus

A model defines the space that can be explored by virtue of the (concrete or abstract) representation choices it reflects, but it does not prescribe the exploration per se, which is the role of the specification. The desk-size model of the 747 facilitates certain kinds of questions and precludes others. These limitations are a direct consequence of the nature of the model, reflecting choices with respect to both focus and mathematical representation. For example, the desk-size 747 does not lend itself to a study of either the safety properties of the airplane's fly-by-wire system or the tensile properties of production-grade materials. The same type of caveat applies to the abstract models used in formal methods. "As with any model, we will have to determine what aspects of reality we deem important and will have to ignore others. We must be quite clear, therefore, on the boundaries of our models" [CS89, p. 94].

4.1.1.3 Expressiveness Versus Analytic Power

There is inevitably a tension between expressiveness and analytic power, as noted in the following quote [CHJ86, p. 9].

> "...in general, the larger the class of systems that can be described, the less is analytically decidable about them. This unfortunate property of mathematics means that great care and mathematical sophistication must be applied to the design of models, especially if a lower level of sophistication is to be expected of the engineers who use them."

Although the author of this quote is talking somewhat pessimistically about engineering models used to compute stresses, mass, friction, and so forth and appears to equate expressiveness and descriptive generality, his observation about the tension between expressiveness and analytic potential is worth noting. In the context of formal methods, expressiveness is typically used to refer to the ability to naturally and effectively characterize a behavior or property of interest. Although generality certainly plays a role, it is not the only hallmark of expressiveness. The analytic potential of a model is crucial in formal methods applications because it is precisely the ability to analyze, that is to calculate and predict, that confers the power and utility of formal methods.

4.1.1.4 Intuitive Versus Nonintuitive Representation

A further consideration can be characterized as naturalness of expression, that is, the extent to which a model should be intuitively similar to the physical object it represents.

Jackson [Jac95, pp. 120-122] cites the example of an electrical network used to model the flow of liquid through a network of pipes. The example is due to Ackoff [Ack62], who terms it an *analogic* model; the wires are *analogous* to the pipes, and the flow of current is *analogous* to the flow of liquid. Ackoff also identifies a class of models that he terms *analytic*, by which he appears to mean that the model embodies an analysis. For example, a set of differential equations describing how prices change is analytic because it expresses the economist's *analysis* of the relevant part of the economy. This is a somewhat different use of the term "analytic" than that of Cohen (above) and most of the literature on formal methods. Although Ackoff's classification is not necessarily advocated here, the notions of analogic and analytic content of models are useful.

4.1.1.5 Accuracy

Finally, it is important to be aware not only of the limitations of models used for formal methods, but also of their accuracy. Just as specification and analysis are constrained by the nature of the model, the ultimate utility and validity of the specification and analysis are limited by the degree to which the model is an accurate representation of the system modeled.

4.1.2 Benefits of Mathematical Models

The advantages conferred by mathematical models are effectively those associated with the more rigorous levels of formal methods, namely

- Mathematical models are more precise than an informal description written in natural language or in quasi-formal notations, such as pseudocode, diagrammatic techniques, and many CASE notations. One aspect of precision is the need to examine and make explicit all underlying assumptions; hence, mathematical models also tend to force a more thorough analysis.

- Mathematical models can be used to calculate and predict the behavior of the system or phenomenon modeled.

- Mathematical models can be analyzed using established methods of mathematical reasoning. The axiomatic method that provides a discipline for proving properties and for deriving and predicting new behaviors from those already known is an example of one such method, in this case drawn from mathematical logic.[3]

Gries and Schneider [GS93, pp. 2–3] use the discovery of the planet Neptune to illustrate some of these benefits of mathematical models. Since it is a particularly nice example of the calculative and predictive power of mathematical models, the story is recounted here. In the early 1800s, it was noted that there were discrepancies between observations of the planet Uranus and the extant mathematical models of planetary motion — largely those formulated by Kepler, Newton, and others beginning in the seventeenth century. The most likely conjecture was that the orbit of Uranus was being affected by an unknown planet. In 1846, after two to three years of feverish manual calculation, motivated in part by a prize offered by the Royal Society of Sciences of Göttingen in Germany, scientists converged on the probable position of the unknown planet. That same year, using telescopes, astronomers discovered Neptune in the position predicted by the models.

---

[3] See Chapter 6.

4.1.3 Mathematical Models for Discrete and Continuous Domains

In an introductory chapter to his classic history of mathematics viewed through the lives and achievements of the great mathematicians, E. T. Bell notes that

> "...from the earliest times, two opposing tendencies, sometimes helping one another, have governed the whole involved development of mathematics. Roughly these are the *discrete* and the *continuous*." ...The discrete struggles to describe all nature and all mathematics atomistically, in terms of distinct, recognizable individual elements, like the bricks in a wall, or the numbers 1,2,3,.... The continuous seeks to apprehend natural phenomena—the course of a planet in its orbit, the flow of a current of electricity, the rise and fall of the tides, and a multitude of other appearances...." [Bel86, p. 13]

This dichotomy is, of course, reflected in the mathematical models used to explore the respective domains. The introductory comments in earlier sections of this chapter have been chosen to apply equally to both discrete and continuous models, thereby emphasizing the commonality between the fundamental role of models in both mathematical domains. Recently, a growing interest in *hybrid systems* — that is, systems composed of continuous components selected, controlled, and supervised by digital components — has led to an integration of discrete and continuous models. The resulting models integrate the differential-difference-type equations used in classical models of continuous physical systems with the mathematical logic and discrete mathematics used in conventional models of digitial systems.[4]

For most of this chapter, the focus will be the discrete domain models typically used in formal methods. While the mathematics exploited in models for discrete domains is generally simpler than that for continuous domain models, it is also less familiar to those with engineering backgrounds. With this in mind, a small example from control theory is presented first. The technical details of the example are not important; the focus here is not on advanced control theoretic methods, but on modeling techniques.

---

[4] Representative papers may be found in the proceedings of several recent workshops, including [GNRR93, AKNS95, AHS96].

4.2 Continuous Domain Modeling

This discussion illustrates the use of continuous mathematics to model an example drawn from spacecraft attitude control. The example was chosen to allow the reader to compare and contrast the continuous model with the discrete model used for the SAFER example, both of which derive from the domain of spacecraft attitude control. In both cases the goals are the same: rigorous description and prediction of behavior. What differs are the character of the underlying mathematics and the techniques used for calculation.

A rigid body or spacecraft in a stable orbit may experience rotational motions that require correction or nulling. A fixed or slowly rotating attitude, pointing the spacecraft at a specific target or in a specific direction, is typically desired. Solving this problem requires a model of rigid body dynamics and, once a control strategy is adopted, a model of the expected behavior under the desired control regime. The mathematical basis for such models is invariably that of differential equations, which offer a well-understood theory to support calculation and prediction.

Following Bryson [Bry94], the rotational motions of a rigid body in space can be modeled as follows: let the angular velocity vector $\vec{\omega}$ be defined with respect to the center of mass and principal body axes, making the products of inertia zero. Let $\vec{\imath}$, $\vec{\jmath}$, $\vec{k}$ be the unit vectors along the x, y, z principal body axes so that

$$\vec{\omega} = p\vec{\imath} + q\vec{\jmath} + r\vec{k} \tag{4.1}$$

Denote by I_x, I_y, I_z the moments of inertia, and by Q_x, Q_y, Q_z the body-axis components of the external torque. The equations of motion describing the body rotations are then given by

$$\begin{aligned} I_x\dot{p} - (I_y - I_z)qr &= Q_x \\ I_y\dot{q} - (I_z - I_x)rp &= Q_y \\ I_z\dot{r} - (I_x - I_y)pq &= Q_z \end{aligned} \tag{4.2}$$

where the time derivative of quantity v is denoted $\dot{v}$. The resultant external torque $\vec{Q}$ includes any intentionally applied torques as well as disturbance torques from sources such as gravitational or magnetic fields.

Consider the problem of achieving attitude hold, that is, applying a time-varying torque to hold a rigid body's rotation at zero or near-zero levels with respect to inertial space. Assume first that any disturbance torques present are small compared to the applied torques and hence may be ignored. This situation exists for "fast attitude control" based on the use of thrusters. Assume further that the mass properties of the rigid body are sufficiently symmetric about the axes so that the axes may be regarded as decoupled and control can be achieved for each axis independently. Finally, assume that appropriate sensors are available to sense both attitude and attitude rate for the axes of interest. For purposes of this discussion, consider a single axis only, the principal y-axis, whose attitude deviation is denoted by θ and attitude rate by $\dot{\theta}$, where $\dot{\theta}$ equals q from equations (4.1) and (4.2).

If the thrusters are proportional, that is, they can be throttled to provide variable amounts of thrust, then attitude control can be achieved using a simple linear control law. The applied torque is derived by feeding back a linear combination of attitude deviation and attitude rate:

$$I_y\ddot{\theta} = Q_y = -D\dot{\theta} - K\theta \tag{4.3}$$

Motion will be stabilized as long as $D > 0$ and $K > 0$.

Proportional gas jets for attitude control are impractical, however, and the more typical method is to use thrusters whose valves are either completely open or completely closed. This leads to what is often termed "bang-bang" control. In pure bang-bang control, thrust is switched between one thruster and its opposing jet, exactly one of which is on at all times. Thus, the control torque has only two values, Q_T and $-Q_T$.

Attitude deviation can be reduced through nonlinear control to nearly zero by applying the torque

$$Q = -Q_T \ \mathrm{sgn}(\theta + \tau\dot{\theta}) \tag{4.4}$$

where

$$\mathrm{sgn}(x) = \begin{cases} 1 & \text{if } x > 0 \\ -1 & \text{otherwise} \end{cases} \tag{4.5}$$

and τ is a constant making $\theta + \tau\dot{\theta}$ a linear switching function, thereby defining a line in the θ-$\dot{\theta}$ phase plane across which thrust reversal occurs. Using this control logic results in the following relationship between Q and the attitude quantities:

$$\dot{\theta}^2 = \frac{2Q}{I}\left(\theta - \theta_0 + \frac{I}{2Q}\dot{\theta}_0^2\right) \tag{4.6}$$

The model predicts a convergence process that drives both θ and $\dot{\theta}$ toward zero, where they will eventually enter a limit cycle surrounding $\theta = \dot{\theta} = 0$.

A further refinement in a practical design would add a "dead zone" around the desired attitude where no thruster firing occurs. Such a scheme is used in the SAFER system described in Appendix C. Hysteresis is typically also incorporated, resulting in control laws with additional nonlinearities. In such cases, the model shown for pure bang-bang control is embellished to capture the more elaborate limit cycle behavior.

The focus now shifts from continuous domain modeling techniques to those of discrete domain modeling.

4.3 Discrete Domain Modeling

This discussion of discrete domain models is intended to be representative rather than exhaustive. To that end, the discussion is framed in terms of four broad classes of discrete domain models: functional, abstract state machine, automata-based, and object-oriented. Of course, there are variants and shadings both within and between these classes, so that the four categories represent a descriptively useful, but somewhat artificial classification.

As the application of techniques from logic and discrete mathematics to problems of interest in computer (hardware and software) systems, formal methods inherently concern computation. By the same token, one of the ways in which formal methods usually differ from traditional uses of logic and discrete mathematics is that they incorporate a model of computation. The model of computation may be built in, that is, implicit, as it is in Hoare logic [Hoa69] and its variants, such as VDM [Jon90] and Z [Spi88, Wor92]—meaning that there is a built-in notion of program state, and a set of constructs for composing operations that affect the state. Or it may be constructed on top of an "ordinary" logic as Hoare logic may be defined within higher-order logic [Gor89]. The advantage of the built-in approach is obvious when the built-in model is appropriate to the task at hand. The advantage of the "constructed" approach is that it is possible to tailor the model to suit the circumstances of a given application. For example, adding concurrency to a sequential Hoare logic is not easy—it generally cannot be done *within* the logic, but requires metalogical adjustments—whereas various models of parallel computation can be encoded in higher-order logic.

One of the key decisions in developing models for formal methods applications is the relevance, if any, of the underlying model of computation, that is, the extent to which the underlying computational paradigm should be explicitly modeled. It is useful to keep this in mind during the discussion of discrete-domain models.

4.3.1 Functional Models[5]

A functional model is one that employs the mathematical notion of function in a *pure* form, often in conjunction with an implicit and very simple computational model. A surprisingly wide variety of algorithms can be adequately described as recursive functions, assuming the most elementary model of computation, namely, the operation of function composition. For example, one of the crucial insights in the specification and analysis of the Byzantine Agreement protocols [Rus92] was the observation that a simple functional model of computation is sufficient, that is, it is not necessary to explicitly model the (inherently complex) distributed computational environments in which these protocols normally execute.[6] For a more concrete example, consider a functional model for a simple synchronous hardware circuit, such as a binary (full) adder that takes three one-bit inputs `x`, `y`, and `c_i` (carry-in) and produces sum and carry-out bits `s` and `c_o`, respectively. In the functional model, a block with several outputs is modeled by several functions, one for each output,[7] and "wiring" is modeled by functional composition. Using this functional model, the binary adder would be then be specified by two functions, one each for `s` and `c_o`:

---

[5] Models for synchronous hardware circuits are used to illustrate many of the ideas in this section. Although these hardware models suggest lower-level, more architectural issues than those discussed elsewhere in this guidebook, the simple hardware models provide more concise, transparent examples of the modeling techniques in question than are typically available with requirements-level specifications.

[6] John Rushby provided this observation, which he credits, in turn, to Bill Young [BY90].

```
s(x, y, c_i) = (x + y + c_i) rem 2
c_o(x, y, c_i) = (x + y + c_i) div 2
```

The relational model, first popularized by Mike Gordon for hardware verification [Gor86], is a variant of the functional model that exploits the more general notion of mathematical relation. In the relational model, a functional block is represented by a single relation on the input and output "wires" that specifies the overall input-output relation. For example, using the relational model, the adder might be specified by the following relation:

```
adder(x, y, c_i, s, c_o) =
    (s = (x + y + c_i) rem 2 AND c_o = (x + y + c_i) div 2)8
```

In the relational model, composition is accomplished by identifying "wires" with variables, conjoining the relations representing the individual blocks, and using existential quantification "to hide" the internal wires.

For example, the implementation of a full adder in terms of half adders and a nand[9] gate can be accomplished by the circuit shown in Figure 4.1. A half adder takes two inputs `a` and `b`, and produces sum (`s`) and (complemented) carry (`c`) bits satisfying

```
half_adder(a, b, s, c): bool = (2 * (1-c) + s = a + b)
```

while a nand gate produces an output (`o`) that is 0 if the sum of its inputs is two, and 1 otherwise:

```
nand(x, y, o): bool = (o = IF x + y = 2 THEN 0 ELSE 1 ENDIF)
```

The "wiring diagram" of Figure 4.1 is then specified by the formula

```
 EXISTS p, q, r :
 half_adder(x, y, p, q) AND half_adder(p, c_i, s, r) AND nand(r, q, c_o)
```

---

[7] In a language such as PVS, that has tuple-types, a single function that produces a tuple, that is, bundle, of values could be used.

[8] A more "requirements" oriented version would be `adder(x, y, c_i, s, c_o) = (2 * c_o + s = x + y + c_i)` (with type constraints restricting all variables to the values 0 and 1).

[9] *Nand* is also known as the *Sheffer stroke* and symbolized as "|". As the name suggests, nand is defined as the negation of the *and* ($\wedge$) operation. Using De Morgan's laws, the $\wedge$ and $\vee$ (*or*) operations, and Boolean variables x and y, nand is defined

$$x \mid y = \neg(x \wedge y) = \neg(\neg(\neg x \vee \neg y)) = \neg x \vee \neg y$$

The nand and nor (*not or*) operations played an important role in logical design because each is *functionally complete*, that is, every switching function can be expressed entirely in terms of either of these two operations.

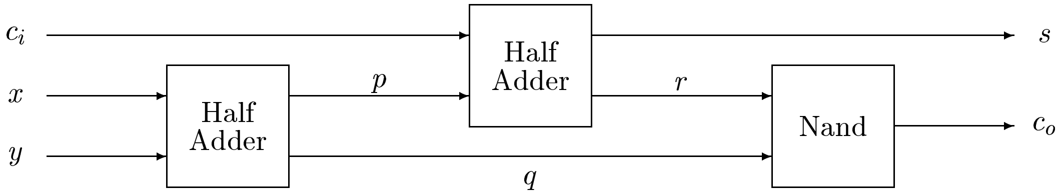


Figure 4.1: Implementation of a Full Adder.

The advantage of the functional approach is that it can lead to very simple and effective theorem proving—basically just term rewriting, and can be "executed" to yield a "rapid prototype." The advantages of the relational approach are that it directly corresponds to wiring diagrams (variables correspond exactly to wires, relations to functional blocks), and that it can cope with feedback loops. It is often possible to combine the methods, as in the first of the relational "adder" examples above, where the conjuncts to the relation correspond directly to the functions of the functional model. The combined approach may additionally involve an explicit representation of state.

4.3.2 Abstract State Machine Models

A state machine model typically consists of an abstract representation of *system state* and a set of operations that manipulate the state to effect a transition from the current to the next state. Figure 4.2 illustrates a basic abstract state machine model. The state machine transition function is a mathematically well-defined function that takes input values and current-state values, and maps them into output and next-state values.

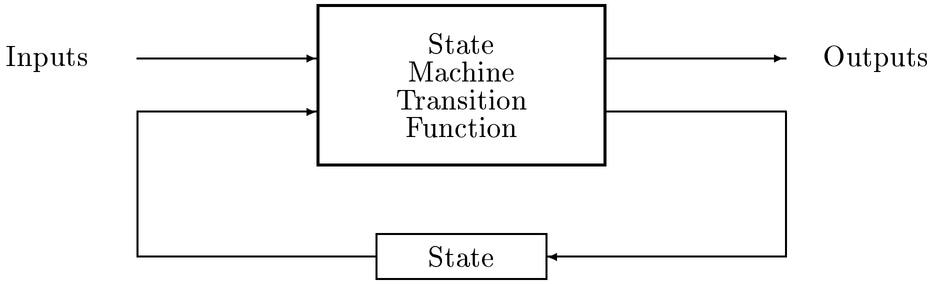


Figure 4.2: Abstract State Machine Model.

Representing each of these values as a vector, this function, M, can be characterized as follows, where I and O are inputs and outputs, respectively, and S is a set of states. Note that this formalization does not explicitly represent the distinction between current- and next-state values.

$$M : I \times S \rightarrow [O \times S]$$

M can be used to capture the functionality of a given system, as well as to formalize abstract properties about system behavior. For example, if sequences $I(n) = < i_1, \ldots, i_n >$ and $O(n) = < o_1, \ldots, o_n >$ denote the flow of inputs and outputs that would occur if the state machine were run for n transitions, then a property about the behavior of M could be expressed as a relation P between $I(n)$ and $O(n)$. Ultimately, it would be possible to formally establish that the property P does indeed follow from the formal specification M.

The A-7 methodology [H+78, Hen80, vS90, Par91, PM91] developed for describing the requirements for control systems illustrates how the state machine model can be specialized to accommodate a particular type of application. In this case, the basic idea is that a control system can be modeled as a control *function* plus a state. The system evolves in time: at each iteration or *frame* it reads the values of certain *monitored variables*, that is, it samples sensors, consults the current values of its *state variables*, and computes a function that yields a pair of results: new values for the state variables and output values for the *control variables*. The dataflow diagram in Figure 4.3 illustrates the basic A-7 model for a system with one monitored variable `x_m`, one control variable `y_c`, and a single state variable `z`, which is denoted `z_s` and `z_f` according to whether it is being read from, or written to, the local state. The purpose of a requirements specification in this context is to specify the box labeled "control."

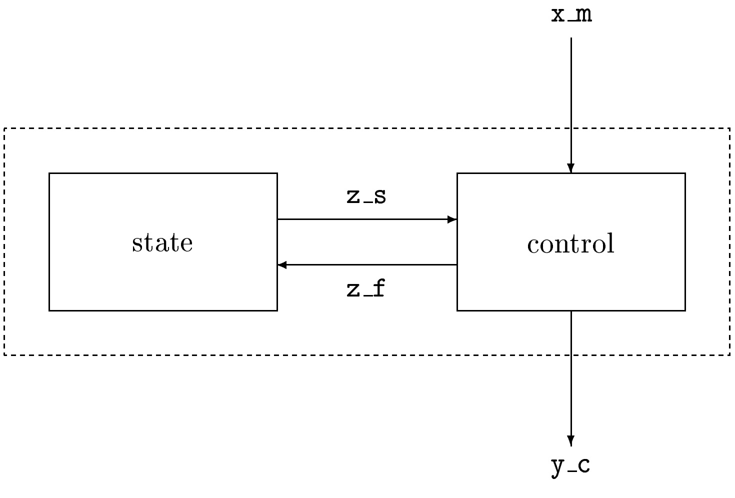


Figure 4.3: A-7 Model of a Simple Control System.

To specify this model of computation explicitly, the variables `x_m` and so on would be modeled as *traces*: functions from time (that is, frame number) to the type of the value concerned. For example, `x_m(t)` is the value of monitored variable `x_m` at time (frame number) `t`. It is then possible to specify how the outputs are computed and how the renaming of `_f` variables to `_s` variables occurs by means of the set of recursive equations:

```
y_c(t) = f(x_m(t), z_s(t))
z_f(t) = g(x_m(t), z_s(t))
z_s(t) = z_f(t-1)
```

where `f` is a function that specifies the computation for the control output and `g` is a function that specifies how the local state value is updated (see Figure 4.4). In general, there will be many monitored, controlled, and state values, and those values themselves can be vectors of values or arbitrary data types.

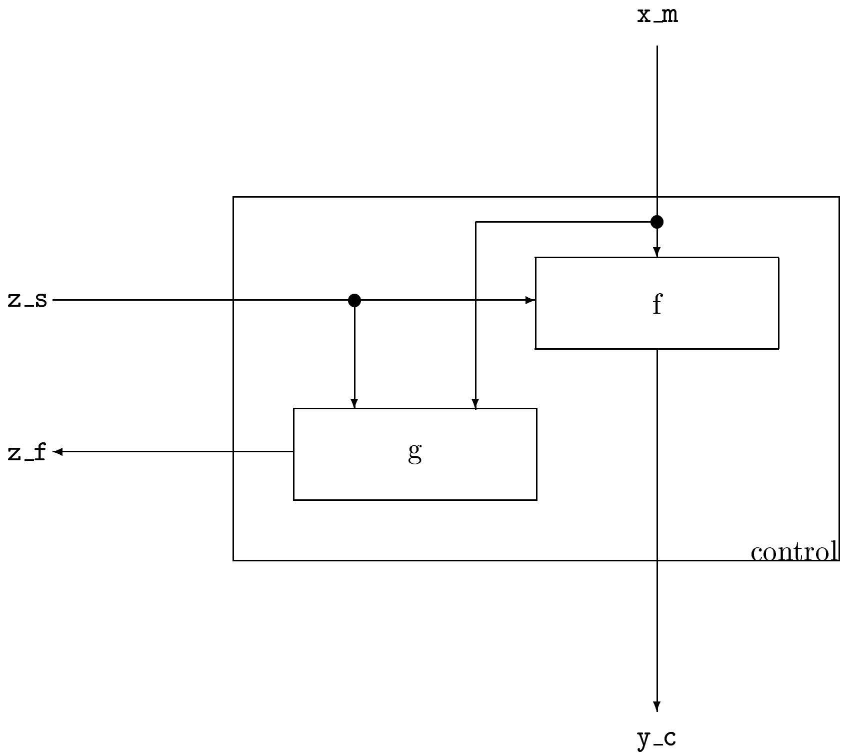


Figure 4.4: State-Update and Actuator Functions within Control System.

On the other hand, if there is no need to reason about the evolution of the system over time, a far simpler representation that uses pure functions on simple values rather than traces may suffice to specify how the "new" values of the various state and output variables are derived in terms of the monitored and "old" values. The conceptual model used to formalize the Jet Select function of the Space Shuttle flight software [NASA93] provides an example of this approach. *Jet-Select* is a low-level Orbit DAP control function that is responsible for selecting which Reaction Control System jets to fire to achieve translational or rotational acceleration in a direction determined by higher-level control calculations or crew input. In the pilot study cited, the behavior of a component, such as the rotation compensation module, would be represented by a function that models the external interface to the function. Note the explicit representation of prior- and next-state values in the signature of the function, f.

$$f : \text{external inputs} \times \text{prior state inputs} \rightarrow [\text{external outputs}, \text{next state outputs}]$$

4.3.3 Automata-Based Models

An automaton is a finite-state transition system consisting of a set of states and a set of state-to-state transitions that occur on input symbols chosen from a given alphabet.

4.3.3.1 $*$-Automata

Automata may be *deterministic*, meaning that there is a unique transition from a given state on a given input, or *nondeterministic*, meaning that there are zero, one, or more such transitions. Formally, a deterministic finite automaton is defined as a 5-tuple $(S, \Sigma, \delta, s_0, F)$, where S is a finite set of states, Σ is a finite input alphabet, s_0 is the initial state, $F \subseteq S$ is the set of final states, and δ, the transition function, maps $S \times \Sigma$ to S. A nondeterministic finite automaton is similarly defined as a 5-tuple, the only difference being that δ is a map from $S \times \Sigma$ to the power set of S, written $\mathcal{P}(S)$. In other words, $\delta(s, a)$ is the set of all states s' such that there is a transition labeled a from s to s'. A thorough introduction to finite automata may be found in [Per90].

Conventional or $*$-automata accept only finite words and can express state invariants, that is, safety properties or properties "at a state", but not eventualities or fairness constraints [Kur94, p. 13].[10]

4.3.3.2 ω-Automata

To accommodate eventualities, it is necessary to use a class of automata that accepts infinite words (sequences), the so-called ω-automata. Like a conventional automaton, an ω-automaton consists of a set of states, an input alphabet, a transition relation, and a distinguished initial state. The difference between the two classes of automata occurs in the definition of *acceptance*. Acceptance for a conventional automaton is defined in terms of a final state. Since the notion of final state is not useful for a class of machines that accepts infinite words, acceptance must be defined in some other way. Various acceptance conditions have been given for ω-automata [CBK90, p. 104], two of which are given below. The definitions that follow are based on a discussion in [CBK90]. A (nondeterministic) ω-automaton is a 5-tuple $(S, \Sigma, \delta, s_0, \mathcal{F})$, where S, Σ, and s_0 are as defined above, $\mathcal{F}$ is an *acceptance condition*, and $\delta : S \times \Sigma \rightarrow \mathcal{P}(S)$ is a transition relation. The automaton is deterministic if for every state $s \in S$ and every $a \in \Sigma$, $|\delta(s, a) \leq 1|$. A comprehensive survey of ω-automata appears in [Tho90].

---

[10] Fairness constraints specify, for example, that certain actions or inactions do not persist indefinitely or that "certain sequential combinations of actions are disallowed" [Kur94, p. 57]. Anticipating the discussion in Section 6.2.1.1, a fairness property can be defined as an LTL property (p) of the type $\mathbf{GF}(p)$. This definition uses CTL* syntax; the definition could also be written using LTL operators.

The following definitions, again taken from [CBK90], are necessary for defining particular instances of $\mathcal{F}$. A *path* in an ω-automaton, M, is an infinite sequence of states $s_0\ s_1\ s_2 \ldots \in S$ that begins in s_0 and has the following property: $\forall i \geq 1, \exists a_i \in \Sigma : \delta(s_i, a_i) \ni s_{i+1}$. A path $s_0\ s_1\ s_2 \ldots \in S^\omega$ in M is a run of an infinite word $a_1 a_2 \ldots \in \Sigma^\omega$ if $\forall i \geq 1 : \delta(s_i, a_i) \ni s_{i+1}$. The *infinitary set* of a sequence $s_0\ s_1\ s_2 \ldots \in S^\omega$, written $inf(s_0 s_1 \ldots)$, is the set of all states that appear infinitely many times in the sequence.

A Büchi automaton M is an ω-automaton where the acceptance condition, $\mathcal{F}$, is defined as follows. $F \subseteq S$ is a set of states (as in the case of a $*$-automaton) and a path p is accepted by M if $inf(p) \cap F \neq \emptyset$. The acceptance condition of a Muller automaton is a set $F \subseteq \mathcal{P}(S)$ of sets of states. A path is accepted by a Muller automaton if $inf(p) \in F$. Other ω-automata that appear in the literature are Rabin, Streett, L, and $\forall -$ automaton. Although acceptance conditions for these automata are not defined here, it is worth noting that "an infinite word is accepted by a Büchi, Muller, Rabin, Streett, or L automaton if it has an accepting run in the automaton. An infinite word is accepted by a $\forall$-automaton if all its possible runs in the automaton are accepted." [CBK90, p. 106]

4.3.3.3 Timed Automata

Timed automata are a generalization of ω-automata and are used to model real-time systems over time. Like ω-automata, timed automata generate (accept) infinite sequences of states. However, timed automata must also satisfy timing requirements and produce (accept) timed state sequences. Timed automata may be given various semantic interpretations, including point-based strictly-monotonic real-time (the original interpretation), interval-based variants, interleaving, fictitious clock, and/or synchronicity [AH91]. An excellent discussion of the theory of timed automata and its application to automatic verification of real-time requirements of finite-state systems may be found in [AD91].

4.3.3.4 Hybrid Automata

Hybrid automata extend finite automata with continuous activities and are used to model systems that incorporate both continuous and digital components. Hybrid automata may be viewed as "a generalization of timed automata in which the behavior of variables is governed in each state by a set of differential equations." [ACHH93] There are various classes of hybrid automata, including linear hybrid automata and hybrid input/output automata. Linear hybrid automata require the rate of change with time to be constant for all variables (although the constant may vary from location to location) and the terms used in invariants, guards, and assignments to be linear.[11] Alur *et al.* [ACHH93] provides a good introduction to hybrid automata and [AH95] describes a symbolic model checker for linear hybrid systems. Hybrid input/output automata (HIOA) focus on the external interface of a modeled hybrid system through distinctions in the state variables — which are partitioned into input, output, and internal variables — and the transition labels — which are similarly partitioned into input, output, and internal actions. Lynch [LSVW96] gives a useful introduction to HIOAs and [AHS96] contains several papers, including [Lyn96], describing the use of HIOAs to model and analyze automated transit systems.

4.3.4 Object-Oriented Models

Object-oriented models represent systems as structured collections of classes and objects with explicit notions of encapsulation, inheritance, and relations between objects. Several informal object-oriented analysis and design methodologies are currently popular, including Booch [Boo91], Coad and Yourdon [CY91a, CY91b], Rumbaugh [RBP$^+$91, RB91], Shlaer and Mellor [SM91], Goldberg [Sys92, RG92] and most recently, Unified Modeling Language (UML) [Rat97]. These methodologies offer a useful and easily assimilated approach for structuring an application based on multiple diagrammatic views of the underlying system. UML, which represents a unification of the Booch, Rumbaugh, and Jacobson methods, employs static structure, use case, sequence, collaboration, state, activity, and implementation diagrams. Rumbaugh's Object Modeling Technique (OMT) [RBP$^+$91] method, which is used in the following example, employs three separate modeling techniques: entity-relationship-type diagrams, state machines or Statecharts [Har87, HN96], and data flow diagrams, yielding a composite model whose components are typically linked rather than integrated or unified.

The following fragment of a design-level OMT representation of a generic fault protection monitor based on a study of the Cassini spacecraft [LA94, AL95] illustrates the use of object-oriented techniques for modeling spacecraft systems. The OMT representation is generic in that it attempts to explicitly document the functionality and attributes shared by all the Cassini fault protection monitors. In the context of spacecraft systems, the term "monitor" refers to software that periodically checks for system-level malfunctions and invokes recovery software as appropriate. There are eighteen monitors in the system-level fault protection onboard the Cassini spacecraft, including eight "over temperature" monitors. The other ten monitors detect loss of commandability

---

[11] A timed automaton is a special case of linear hybrid automaton in which each continuously changing variable is an accurate clock whose rate of change with time is 1. In a timed automaton, all terms involved in assignments are constants and all invariants and guards compare clock values with constants [ACHH93].

(uplink), loss of telemetry (downlink), heartbeat loss (that is, loss of communication between computers), overpressure, undervoltage, and other selected failures.

The OMT approach provides three viewpoints: the object model, the functional model, and the dynamic model. Figures 4.5, 4.6, and 4.7 illustrate these three models for the Cassini fault monitor at the design level.

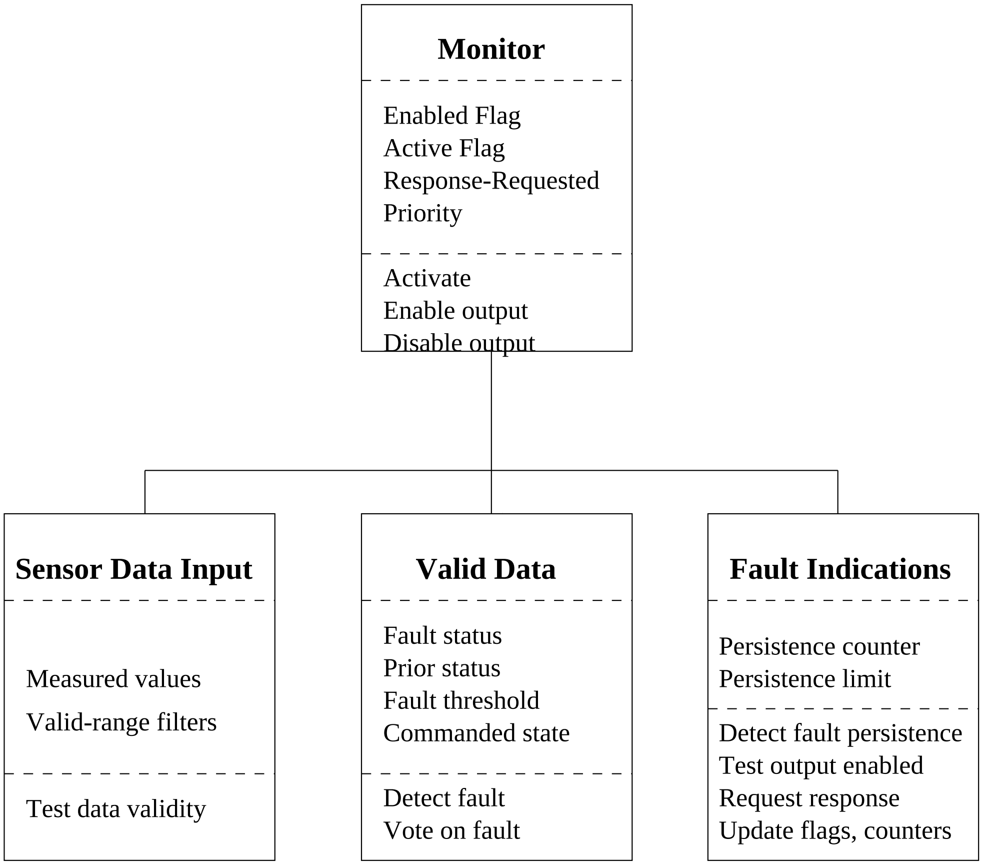

Monitor
Enabled Flag
Active Flag
Response-Requested
Priority
Activate
Enable output
Disable output
Sensor Data Input
Measured values
Valid-range filters
Test data validity
Valid Data
Fault status
Prior status
Fault threshold
Commanded state
Detect fault
Vote on fault
Fault Indications
Persistence counter
Persistence limit
Detect fault persistence
Test output enabled
Request response
Update flags, counters

Figure 4.5: Object Model of Cassini Generic Fault Protection Monitor.

Figure 4.5 reproduces the object model, a static representation of the system that reflects four attributes and three operations that define the monitor class (activate, enable output, and disable output). The class is further decomposed into three object classes: sensor data, valid data, and fault indicators. The attributes and operations for these three classes define the interfaces between the monitor class and the rest of the system.

Figure 4.6 reproduces the functional model, which represents the computation that occurs within a system and is presented as a series of data flow diagrams. The top-level diagram documents the interfaces between the fault protection manager and the

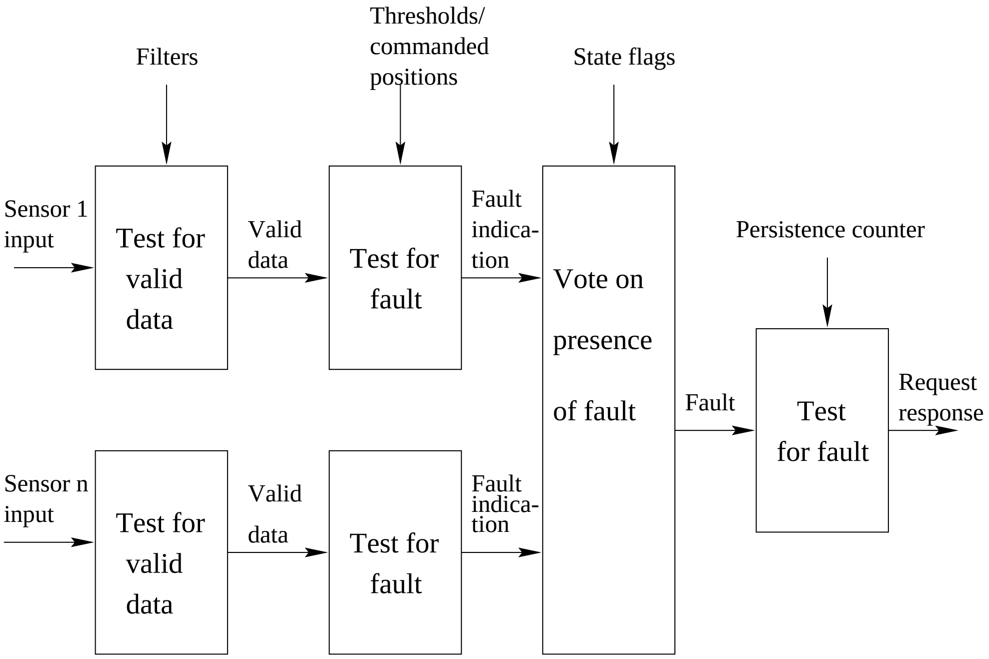

Filters
Thresholds/ commanded positions
State flags
Sensor 1 input
Test for valid data
Valid data
Test for fault
Fault indica- tion
Vote on presence of fault
Persistence counter
Fault
Test for fault
Request response
Sensor n input
Test for valid data
Valid data
Test for fault
Fault indica- tion

Figure 4.6: Functional Model of Cassini Generic Fault Protection Monitor.

monitor. The manager activates the monitor and processes the monitor's request for a fault response. The monitor receives data from the hardware sensors ("measured state"), from the "commanded state" that is stored in memory, and from the updates to the state made by previous executions of the monitor itself, and uses the information to determine an appropriate fault response.

Figure 4.7 reproduces the dynamic model that specifies the flow of control, interactions, and sequencing of operations. These dynamic aspects are modeled in terms of events and states using standard state diagrams (that is, graphical representations of finite state machines). The behavior of the Cassini fault protection monitors is highly sequential. The state transition model provides a clear and intuitively straightforward representation of the typical six-state sequence followed by an active monitor in the presence of a fault that requires a recovery response.

While the types of informal object-oriented models illustrated here have considerable utility, their usefulness in the context of formal methods is limited because they do not have an underlying mathematical basis and therefore lack a precise semantics and the ability to support formal reasoning. More general caveats expressed in regard to some or all of these informal object-oriented methods include the following [Jac95, p. 137]: (1) objects belong to fixed classes—the rigidity of these class structures precludes transition or metamorphosis of objects; (2) objects typically inherit properties and behavior from a single class at the next hierarchical level; this notion of single inheritance precludes many

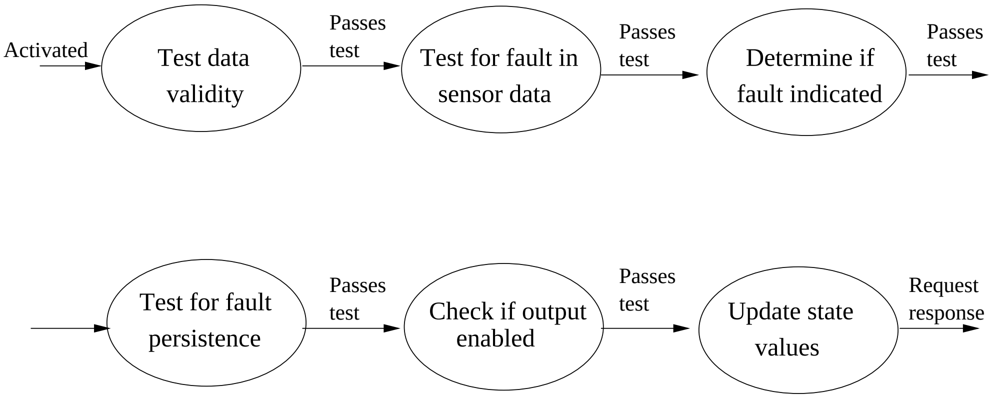

Activated
Test data validity
Passes test
Test for fault in sensor data
Passes test
Determine if fault indicated
Passes test
Test for fault persistence
Passes test
Check if output enabled
Passes test
Update state values
Request response

Figure 4.7: Dynamic Model of Cassini Generic Fault Protection Monitor.

naturally occurring inheritance patterns involving shared and multiple inheritance; (3) objects are inherently reactive and typically cannot initiate activity of any kind. Although these three caveats are now addressed in many object-oriented programming languages, for example, through multiple inheritance, dynamic object classification, and concurrency, the popular methodologies that support the earlier stages of development do not typically address these issues. A fourth caveat is that the lack of integration in composite models often makes it difficult to reason effectively about system behavior.

Historically, object-oriented ideas evolved from the notions of classes and objects in Simula 67. In the following quote, Ole-Johan Dahl discusses this evolution in the context of formal techniques.

> "Object orientation, as it appears in Simula 67, was motivated by two main concerns: To achieve good structural correspondence between discrete event simulation programs and the systems being modelled. And to provide language mechanisms for the construction of reusable program components while maintaining good computer efficiency.... Object orientation has proved to be a successful structuring strategy also outside the area of simulation. This is due to the fact that objects are useful general purpose tools for *concept modelling*, and can lead to better program decomposition in general, as well as new kinds of reusable program components. It is worth noticing that the class concept of Simula 67 is used to represent "modules" and "packages" as well as object classes." [Dah90]

Object-oriented ideas share this ancestry with algebraic specification; the classes of objects and "prefixing" central to Simula 67 ultimately led to object-oriented programming languages and to the theory of algebraic specifications [Bre91]. Algebraic specifications treat data structures and program development concepts, such as refinement, in an axiomatic logical style and use high-level descriptions of data types known as *abstract data types*. Abstract data types are manipulated by similarly high-level operations that are specified in terms of properties, thereby avoiding implementation-dependent data representations. As Abadi and Cardelli note in their book on the (formal) foundations of object-oriented programming languages [AC96, p. 8], "...data abstraction alone is sometimes taken as the essence of object orientation." This historical connection is of interest because the frameworks of algebraic specification and of object-oriented programming languages have each contributed to ongoing attempts to provide a mathematical basis for the concepts underlying object-oriented models.[12] This research has taken many directions, including those summarized below. In keeping with the focus of this guidebook, the examples included in this discussion suggest the variety of the work in this area, but are by no means exhaustive.

One approach is to take a model generated by one of the informal object-oriented methodologies and formalize it using a novel or existing formal description technique. For example, Moreira and Clark [MC94] describe a technique for producing a formal object-oriented analysis model that integrates the static, dynamic, and functional properties of an object-oriented model created using one of the informal object-oriented methodologies.[13] The formal model uses LOTOS (Language of Temporal Ordering Specification) [ISO88], which has a precise mathematical semantics and represents the system as a set of communicating concurrent objects.[14] An object is represented as the instantiation of a LOTOS process, and communication among objects takes the form of message passing, which is modeled by objects synchronizing on an event during which information may be exchanged. In this approach, the dynamic aspects of a class template are modeled as a LOTOS process and the static properties as abstract data types.

Another approach is to take notation from one of the informal methodologies and formalize it, thereby providing a formal semantics for the informal notation. For example, Hayes and Coleman [HC91] use Objectcharts[15] [CHB92] and a derivative of VDM [Jon90] to provide a coherent set of formal models corresponding to the models generated by a subset of OMT. Briefly, Hayes and Coleman introduce an *object structure model*, linking the formal representations of the informal OMT models (object, dynamic and functional) to provide traceability and consistency checking. The informal OMT functional model is replaced by VDM-style pre-post condition specifications over the object structure model, the informal dynamic model is formalized using Objectcharts, and the object model uses the formalized entity-relationship notation described in [FN86]. There has also been work integrating formal and object-oriented methods using VDM++ and OMT [LG96]. VDM++ is an object-oriented extension of VDM designed to support parallel and real-time specification.

---

[12] See, for example, recent proceedings from conferences such as ECOOP (European Conference on Object-Oriented Programming [TP94, Olt95]) and OOPSLA (Object-Oriented Programming Systems, Languages, and Applications) [ACM94].

[13] [MC94] actually describe a Rigorous Object-Oriented Analysis (ROOA) method that combines object-oriented analysis and formal description techniques. This discussion focuses only on their modeling approach.

[14] That is, a set of communicating processes. The approach is based on process algebra, drawing on elements from CCS [Mil89] and from CSP [Hoa85].

[15] An Objectchart is an extended form of Statechart [Har87, HN96] used to specify object classes.

Ongoing work at the Michigan State University Software Engineering Research Group [BC94,CWB94] is yet another variant on this approach. Their prototype system uses algebraic specifications to formalize a subset of the OMT object-modeling notation appropriate for modeling requirements. Again, the formalization is based on the straightforward mapping between object-oriented software concepts and abstract data types.[16]

The CoRE method [FBWK92] for specifying real-time requirements provides a further example of the coherent integration of object-oriented and formal models. CoRE is an amalgam of the CASE Real-Time Method (which is itself an amalgam of Real-Time Structured Analysis [WM85] and object-oriented concepts) and the four-variable model [vS90, vSPM93] developed by Parnas and his colleagues. CoRE interprets the three basic structural elements of the CASE Real-Time method: information, process, and behavior pattern, in terms of object-oriented concepts. Processes correspond to object classes and interprocess connections to interactions between objects. The state machines used to encode the behavior-pattern view are partitioned to correspond to the states of an object class. The formal model underlying object definition and decomposition is based on the standard mathematical model of embedded-system behavior used by the four-variable method. The resulting amalgam retains the graphical notation and notions of abstraction, encapsulation, separation of concerns, and nonalgorithmic specification associated with object-oriented approaches, within a mathematically well-defined model contributed by the four-variable method.

There have also been formalizations in Z of the three OMT notations [Spi88,Wor92], as well as object-oriented extensions to Z. The collection of papers in [SBC92] contains accounts of both approaches, including a summary of Hall's object-oriented Z specification style, which is also described in Hall [Hal90].

4.4 A Model for the SAFER Avionics Controller

The SAFER avionics controller described in Section 2.6 exhibits several characteristics that strongly influence the choice of a model for its formalization. The basic functionality of the controller requires a representation that captures the mapping from input and sensor values to outputs. The model must also be able to capture the dependency of current events on prior events, necessitating the use of a state- or trace-based model, or other representation with similar facility for preserving values from one "cycle" to another. The fact that the controller maintains and updates its own internal status, including Hand Controller Module (HCM) display and Automatic Attitude Hold (AAH) status, provides additional motivation for an explicit representation of state. In fact, the SAFER avionics controller provides a nice illustration of a system that can be quite naturally modeled as a state machine (see Section 4.3.2), that is, as a model consisting of a system state and a transition function that maps inputs and current-state values into outputs and next-state values. Arguably, a variant of the basic state machine model, such as the A-7 [H^{+}78,Hen80,vS90,Par91,PM91], which is specialized for control systems, would provide a representation that differentiates inputs, outputs, and state values by explicitly identifying monitored, control, and state variables (see Figure 4.3). Although the differences between these two models are small, the choice between a basic state machine model and a specialized state machine model illustrates the type of trade-off that typically enters into modeling decisions. In this case, the trade-off is the relative simplicity of the basic state machine model versus the additional expressiveness of the specialized A-7 model, where finer-grained distinctions among variables potentially provide a clearer mapping between informal description, requirements, and the formal specification. On the other hand, the level of description and the (primarily) pedagogical role of the SAFER example motivate the use of the simpler model presented here. Nevertheless, the reader is encouraged to consider the similarities between the basic state machine model developed here and A-7-type models, in particular the notion of the state transition function defined as a control function with monitored (that is, sensor) and state variables as input and control and state variables as output.

---

[16] The graphical environment prototype generates Larch specifications [CWB94]. Although current versions of Larch are not inherently algebraic, the implementation cited supports only algebraic languages although it is general enough to accommodate most algebraic languages that have a well-defined grammar. It appears that "object model" has replaced the previously used phrase *analysis object schemata* (*a-schemata*) in recent publications [BC95b].

A final consideration concerns the representation of time. Since the basic functionality of the controller can be captured within a single frame or cycle, there is no need to reason about the behavior or evolution of the system over time or to introduce the additional complexity required for an explicit representation of time. The trade-off here is the simplicity of the model versus a loss of analytical power. Without an explicit representation of time, there is no way to explore certain types of properties, including safety and liveness properties that establish (roughly) that nothing bad ever happens and something good eventually happens, respectively. For example, without an explicit representation of time, it would be impossible to demonstrate that an HCM translation (rotation) command eventually results in thruster selection.[17] Although the models presented in this chapter do not incorporate a notion of time, a time- or trace-based model could be added, as needed, on top of the state-based model presented here.

Having identified the underlying model as a basic state machine, the next step is to define the control (transition) function. The transition function for the top-level controller model is comprised of functions representing its constituent modules and assemblies. Of interest here are the AAH and thruster selection functions. Thruster selection maps HCM and AAH commands into an integrated six degree-of-freedom command that determines the corresponding (thruster) actuator commands. This two-phase functionality can be modeled simply as the composition of the two functions, roughly

---

[17] Whether the thruster selection is correct with respect to the thruster select logic is an important property, but not a liveness issue.

$$selected\_actuators \circ integrated\_command$$

The AAH model cannot be so simply discharged, because the automatic attitude hold capability maintains internal state information to implement the AAH control law and to track whether the AAH is engaged or disengaged and which, if any, of the three rotational axes are under AAH control. AAH control law is implemented in terms of a complex feedback loop that monitors inertial reference unit (IRU) angular rate sensors and temperature sensors (one for each of the three rate sensors), and generates rotation commands. Although this account is necessarily simplified, it suggests a fairly complex control system with clearly differentiated variable types and a well-defined internal state. The rationale for considering an A-7-type interpretation of a basic state machine model for the top-level avionics controller applies equally to the AAH. The AAH state machine model is shown in Figure 4.8.

A closer look at the AAH button transition function further illustrates the type of issues that invariably arise in developing models for formal specification. The state transition diagram for this function shown in Figure 4.9 represents the single-click, double-click engagement protocol described in Section 2.6, where nodes represent AAH states and arcs represent the two button positions (up or down) and the two operative constraints (timeout or all three rotational axes removed from AAH control).[18]

For example, if the AAH is engaged and the AAH pushbutton switch is depressed, the AAH enters a state ("pressed once") that is exited only when the pushbutton is released, at which point the AAH transitions to a state that may be exited in one of two ways: either the 3-axes-off constraint becomes satisfied and the AAH is disengaged or the pushbutton is depressed for a second time and the AAH enters a twilight state ("pressed twice") prior to button release and disengagement. Several interesting questions arise with respect to this model, largely because of undocumented behaviors. For example, the Operations Manual [SAFER94a] doesn't mention the case represented by the two 3-axes-off arcs, where the axis-by-axis disabling (resulting from explicit rotation commands issued while AAH was engaged) effectively disengages the AAH. The two options are either to leave AAH nominally active with all three rotational axes off or to explicitly inactivate the AAH. The AAH model presented here reflects the second option, which is more straightforward and avoids the possibility of misleading a crewmember into thinking that the AAH is engaged when in fact all three axes have been disabled. There are also modeling issues, including those surrounding the representation of the 3-axes-off transitions. In the model diagrammed above, the 3-axes-off transition emanates only from the "AAH on" and "AAH closing" states, although logically, it can be argued that 3-axes-off transitions should also emanate from the "pressed-once" and "pressed-twice" states. In other words, the model should explicitly reflect that fact that if AAH is engaged and all three axes have been disabled, AAH is terminated. The rationale for the given model is that the behavior of the resulting system is cleaner if the "AAH off" state is entered only after the pushbutton switch is released ("up"). Otherwise the button would be depressed and cause an immediate transition to "AAH started" on the next pass. Similarly, although it is arguably preferable to omit the 3-axes-off transition from "AAH closing" and allow the double click to complete, if the crewmember forgets the second click, another ill-defined situation results.

---

[18] The diagram actually represents a combination of pushbutton and implied events. For example, although the 3-axes-off transition reflects one or more previous HCM commands, it does not represent an explicit pushbutton event, such as AAH enable/disable.

So far, the discussion has focused on modeling SAFER's functionality rather than its physical components. Although many of SAFER's physical features fall below the level of abstraction chosen for the formalization, certain features such as the thrusters must be modeled. SAFER has 24 thrusters arranged in four groups (quadrants) of six thrusters each. Consistent with the intermediate level of detail chosen to make the guidebook example easier to understand, the thrusters are modeled by enumerating each of the 24 thrusters by name and providing a function that maps a thruster name to a full thruster designator. The thruster designator is a triple consisting of elements that represent the direction of acceleration yielded by firing the thruster, its quadrant, and its physical location as shown in Figure C.3. For example, thruster F1 would be mapped to the designator `(FD, 1, RR)` and thruster L3R would be mapped to the designator `(LT, 3, RR)`. Possible values for the three designator components are as follows:

- Direction: up, down, back, forward, left, right
- Quadrant: 1, 2, 3, 4
- Location: forward, rear

It is instructive to consider a more abstract model of the SAFER thrusters. For example, a considerably higher-level model might simply provide primitive (uninterpreted) elements called thrusters, some of which accelerate up, others down, back, forward, right, or left. These distinctions are disjoint, that is, a thruster accelerates in exactly one direction and there are no other kinds of accelerations. The exact number of thrusters and their physical positions with respect to quadrant and location are irrelevant at this level of abstraction, although it would certainly be possible to specify an upper bound on the number of thrusters. The advantage of this highly abstract model is that it is not obscured by (arguably irrelevant) detail and it is general enough to be applicable to new designs or future modifications.

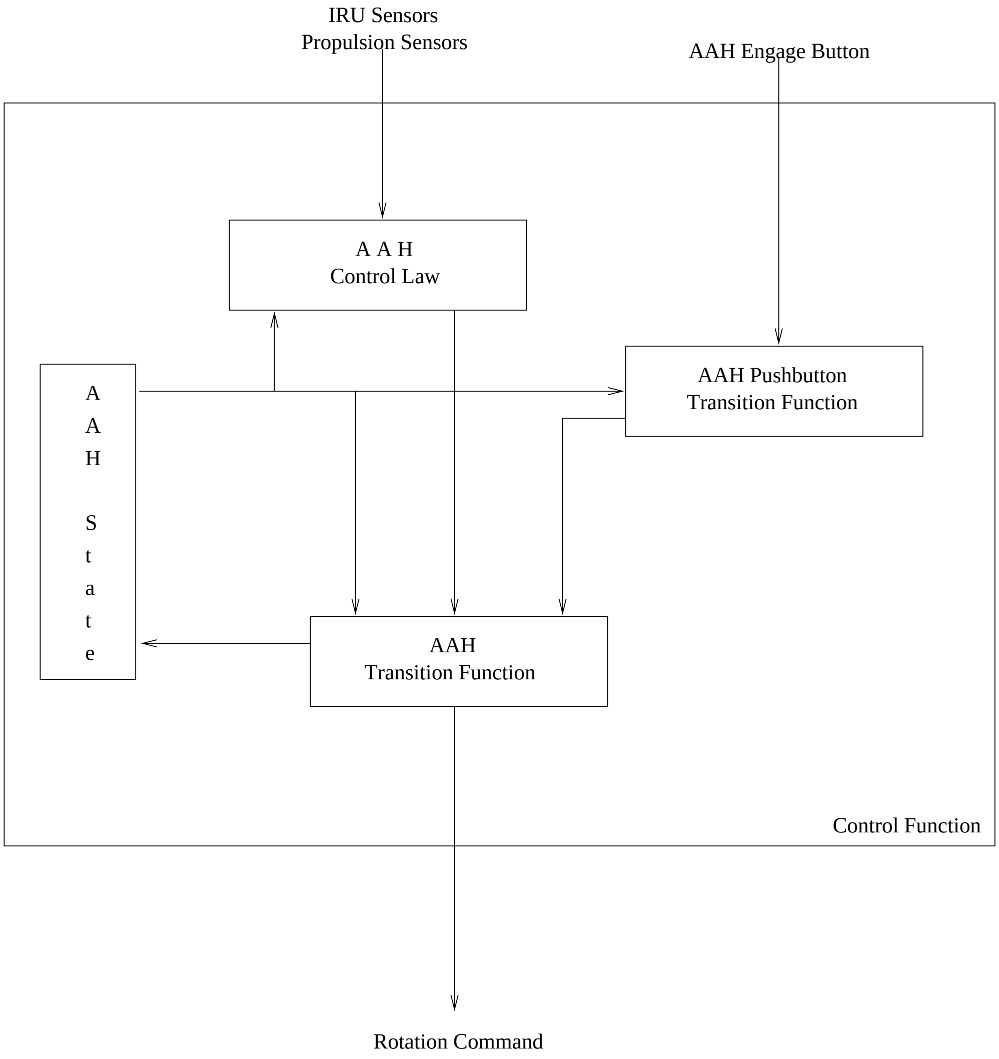
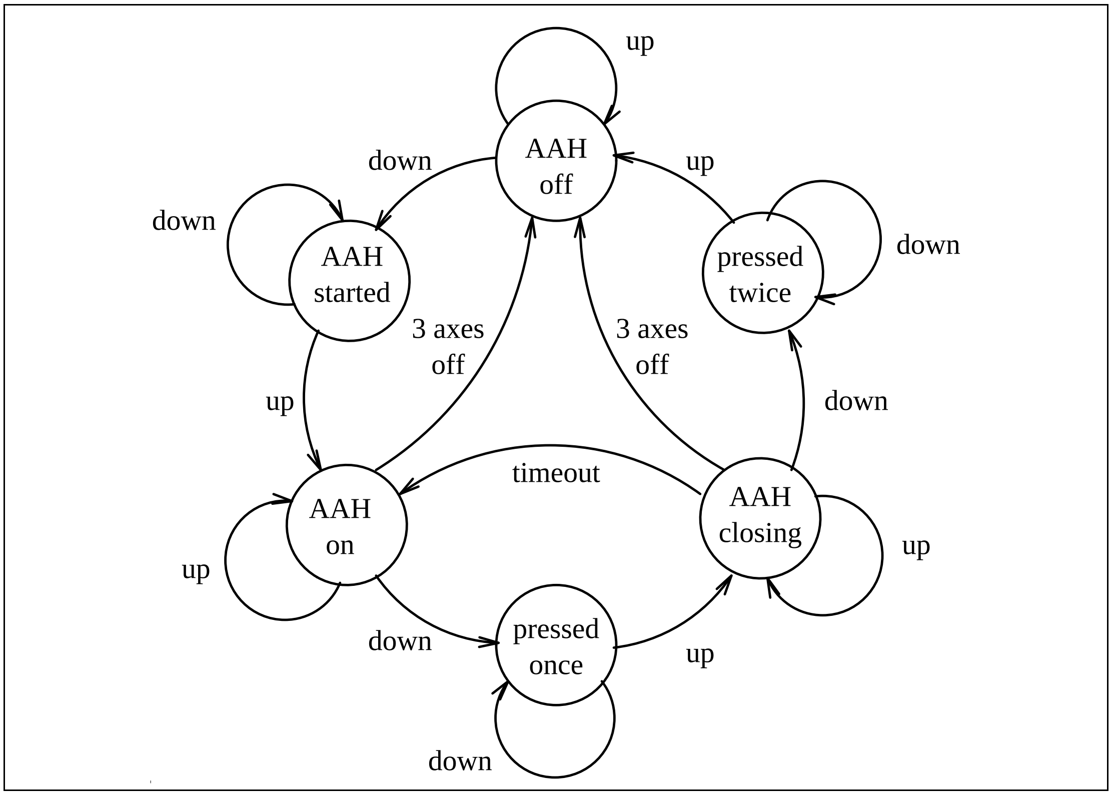

IRU Sensors
Propulsion Sensors
AAH Engage Button
A A H
Control Law
AAH Pushbutton
Transition Function
A
A
H
S
t
a
t
e
AAH
Transition Function
Control Function
Rotation Command

Figure 4.8: AAH Control System State-Update and Actuator Functions.



up
AAH off
down
up
down
AAH started
pressed twice
down
3 axes off
3 axes off
up
down
timeout
AAH on
AAH closing
up
up
down
pressed once
up
down

Figure 4.9: Labeled AAH Pushbutton State Transition Diagram.

Chapter 5

Formal Specification

A formal specification is a characterization of a planned or existing system expressed in a formal language. The characterization typically consists of a collection of axioms and definitions whose meaning and consequences are determined by the precise mathematical basis of the formal language and its rules of inference. In this context, "consequences" denotes all the formulas that can be derived from the axioms and definitions using formal deduction (as prescribed by the inference rules). These derivations are also referred to as proofs, and the set of formulas constitutes the theory defined by the specification. The act of formalizing a specification does not necessarily make it relevant, coherent, or true. There are several ways to increase the certainty that a specification expresses the intentions of its author and that what it says is true, including—in ascending order of rigor—parsing, typechecking, animating, or executing all or part of the specification, well-formedness checking for definitions[1], demonstrating consistency for axiomatic specifications[2], and developing and proving theorems entailed by the specification. Of course, there is no way to completely guarantee that a formal specification is correct or accurately represents reality; the various checks and tools cited here can reduce, but never totally eliminate, the possibility of human error. Nevertheless, there are very real benefits to be gained from formal specification, benefits that are not diminished by the impossibility of definitive correctness.

This chapter focuses exclusively on formal specification, leaving issues of formal analysis and proof to Chapter 6. The discussion covers specification languages and styles, as well as the checks and tools mentioned above with the exception of theorem proving, which, as already noted, is deferred until Chapter 6. The discussion also touches on the utility of formal specification in the absence of formal proof and continues the ongoing example with a partial specification of SAFER, using the model developed at the end of Chapter 4.

---

[1] That is, assuring conservative extension; see Section 5.1.2.9 for a discussion of this and related topics.

[2] For example, exhibiting a model; see Section 6.1.1.

5.1 Formal Specification Languages

A formal language consists of a collection of symbols drawn from an alphabet and a set of syntactic rules that govern which combinations of symbols constitute valid expressions in the language. In purest form, a formal language and the rules for manipulating it are referred to as a (mathematical) logic. The propositional and predicate calculi are examples of this type of formal system. Although some formal specification languages use pure logics, many enrich the underlying logic with modern programming language concepts and constructs such as type systems, encapsulation, and parameterization, thereby increasing the expressiveness of the formal language while retaining the precise semantics of the underlying logic. As these remarks suggest, the distinction between a specification language and a programming language is somewhat blurred. The same can be said for their respective artifacts. Although a program can be viewed as a specification, a specification is typically not a program and often contains such noncomputational constituents as high-level constructs and logical elements (e.g., quantifiers). The basic difference is one of focus: a program specifies completely how something is to be computed, whereas a specification expresses constraints on what is to be computed. As a result, a specification may be partial or "incomplete" and still be meaningful, but an incomplete program is generally not executable [Win90, p. 8] [OSR93a, p. 2].

There is a wide variety of formal specification languages, far too many to be considered here. Rather than focus on a representative sample of these languages, the discussion concentrates instead on general characteristics and features of specification languages, the rationale being that discussion of foundational issues, general features of, and desiderata for formal specification languages will provide the reader with background and access to a wide range of formal specification languages. Although mechanized support for formal systems is not discussed, one of the additional benefits of a high degree of formalization is that specifications written in a formal language are amenable to mechanical analysis and manipulation. Most formal specification languages are supported by mechanized syntax analysis tools, and many also enjoy some level of mechanized semantic analysis, as well as deductive apparatus in the form of theorem provers and proof checkers. Although most systems are designed around a particular specification language and its proof rules, there are also generic systems such as Isabelle [Pau88] that support a variety of logics and notations. Volume I of this guidebook [NASA-95a] includes an extensive list of formal methods tools, as well as a description of approximately 15 of the most widely used of these systems.

5.1.1 Foundations[3]

As noted earlier, a formal specification language is grounded in a mathematical logic. There are, of course, a wide variety of logics: simple propositional logics (either classical or intuitionistic), equational logics, quantificational logics, model and temporal logics, set theory, and higher-order logic, although this by no means exhausts the possibilities. These and other logics were developed by mathematicians to explore issues of concern to them. As Rushby [Rus93b, p.214] notes:

---

[3] The material in this section is based largely on a discussion in [Rus93b].

> "Initially, those concerns were to provide a minimal and self-evident foundation for mathematics; later, technical questions about logic itself became important. For these reasons, much of mathematical logic is set up for *meta*mathematical purposes: to show that certain elementary concepts allow some parts of mathematics to be formalized *in principle*, and to support (relatively) simple proofs of properties such as soundness and completeness."

On the other hand, formal specification languages are developed primarily to be used, that is, to formalize requirements, designs, algorithms, and programs and to provide an efficient and effective basis for reasoning about these artifacts and their properties. Predictably, the languages developed by mathematicians are not necessarily well-suited to the needs of those engaged in formal specification and analysis. This is particularly true when mechanization of specification and analysis is considered.

Although there are specialized uses for some of the logics mentioned above—for example, a propositional or modal logic can provide a basis for efficient determination of certain properties of finite state machines—the logical foundation for an expressive, general-purpose specification language is generally either axiomatic set theory or higher-order logic. Historically, these approaches were developed in response to *Russell's Paradox*, which exposed a fundamental inconsistency in Frege's logical system on the eve of its publication and frustrated Frege's attempts to provide a consistent foundation for the whole of mathematics.[4] Axiomatic set theory avoids contradictions by restricting the rules for forming sets—basically, new sets may be constructed only from existing sets. There are different axiomatizations, characterizing distinct set-theories; the best known of these is called Zermelo-Fraenkel or simply ZF, after its founders [FBHL84, Hal84]. ZF contains eight axioms, all of which express simple, intuitive truths about sets. ZF set theory provides the logical framework for several well-known specification languages, including Z [Spi88] and Verdi, the language of the Eves system [CKM$^+$91]. The main issues surrounding the use of axiomatic set theory as the basis for a specification language are unconstrained expressiveness, the difficulty of providing semantic checking for an inherently untyped system, and the challenge of providing efficient theorem proving for a system in which functions are inherently partial.

In the context of logics, the suffix "-order" refers to the elements over which the logic permits quantification. The standard progression is as follows. The propositional calculus does not allow quantification and is effectively "zero-order." The predicate calculus, which allows quantification over individuals, is referred to as "first-order" logic. Similarly, "second-order" logic provides quantification over functions and predicates on individuals, and "third-order" provides quantification over functions and predicates on functions. The enumeration continues up to ω-order, which allows quantification over arbitrary types and is therefore generally equated with type theory or higher-order logic.

---

[4] Actually, Frege, Cantor, and Dedekind were greatly disillusioned by the contradictions that plagued their set theoretical foundation for the real numbers, continuity, and the infinite and quit the field, leaving the development of a consistent set theory to others. The intellectual history of this period, as well as the mathematics, is fascinating, but well beyond the scope of the guidebook. Rushby [Rus93b, pp. 254-5] offers a brief sketch of the issues based on material in [Hat82, Lev79, FBHL84, Sho78a, And86, BP83, vBD83, Haz83]. The last chapter of Bell [Bel86] provides an equally brief history of the personalities as well as the mathematics.

Axiomatic set theory assumes a flat universe; individuals, sets, sets of sets, ..., are undifferentiated with respect to quantification, which is inherently first-order. Furthermore, axiomatic set theory admits only two predicates: ($\in$ and $=$).[5] In type theory, the universe is ordered with respect to a type hierarchy and quantification must respect the type distinctions. In other words, quantifiers apply to typed elements and the type distinctions must be consistently maintained throughout the scope of the quantifier.

In highly simplified terms, simple type theory avoids the logical paradoxes by observing a strict type discipline that prevents paradoxical circular constructions (also called *impredicative* definitions).[6] The simple theory of types has been used as the basis for several formal methods and theorem proving systems, including HOL [GM93], PVS [ORSvH95], and TPS [AINP88]. As a foundation for formal specification languages, type theory offers several advantages, such as strong, mechanized *typechecking* that confers early and effective error detection; expressive power of quantification and higher-order constructions; and the potential for mechanized theorem proving facilitated by the total functions that typically underlie simple type theory.

5.1.2 Features

The previous discussion of mathematical foundations suggests that the mathematical basis of a specification language figures importantly in determining such features as expressiveness and mechanizability. This section briefly considers expressiveness and other basic features of specification languages. As noted previously, mechanization issues generally lie outside the scope of this guidebook, which is aimed at the practitioner rather than the provider of formal methods tools or systems.[7]

---

[5] Although ZF reconstructs functions and predicates within set theory as sets of pairs, this set theoretic approach is arguably less suitable for formal methods because it tends to be less expressive and less easily mechanized.

[6] The account presented here is very sketchy. Rushby [Rus93b, pp. 270-278] presents a somewhat more thorough discussion, based on material in Andrews [And86], Hatcher [Hat82], Benacerraf and Putnam [BP83], van Bentham and Doets [vBD83], and Hazen [Haz83]. Barwise and Etchemendy [BE87] have published a very readable analysis of the semantic paradox known as "The Liar," using an extension of ZF set theory.

[7] See, for example, Rushby [Rus96], which touches on the implications of specification language design for automated deduction while advocating an integrated approach to automated deduction and formal methods.

5.1.2.1 Explicit Semantics

To provide a basis for mathematically well-defined, credible specifications, as well as a standard framework for understanding the specifications, a specification language must itself have a mathematically secure basis. Ideally, the language should have a complete formal semantics, although languages built on standard logics without significant extensions typically don't have or need a completely formal semantics. On the other hand, specification languages that are not based on standard logics or that employ novel or nonstandard constructions should provide a formal semantics that has undergone some form of peer review or collegial scrutiny. Spivey's formal semantics for Z [Spi88] is an example of this kind of formal semantic account.

5.1.2.2 Expressiveness

As noted earlier, first-order predicate calculus with equality is generally considered the minimum foundation for a reasonably expressive specification language. On the other hand, more restricted bases may be appropriate for particular applications and more powerful bases (such as set theory and higher-order logic) are desirable for most applications. Of course, there are several dimensions to the notion of expressiveness, including flexibility, versatility, and convenience and economy of expression. Some of these derive from other features; for example, a rich type system facilitates more succinct specification since much of the specification can be embedded in the types, as illustrated in the two versions of the claim, *c*1 (below), that the sum of two even integers is even. The property of being an even integer is characterized by the predicate *even*?.

```
x, y: VAR int
c1: CLAIM even?(x) AND even?(y) IMPLIES even?(x + y)
```

Alternatively, the constraint may be embedded in the type, so that variables x and y are declared to be elements of the type consisting (only) of even integers.

```
x, y: VAR {z: int | even?(z)}
c1: CLAIM even?(x + y)
```

Similarly, the availability of familiar programming language datatypes and constructions confers considerable convenience and clarity when dealing with such structures as arrays, records, lists, and sequences. There are also trade-offs; for example, in the case of executable specification languages, finiteness constraints imposed by executability can compromise expressiveness.

5.1.2.3 Programming Language Datatypes and Constructions

Most specification languages support at least some of the familiar programming language datatypes, such as records, tuples, and enumerations, as well as constructions that update these structured types.[8] Some also support abstract data types, including "shell" mechanisms for introducing recursively defined abstract data types, such as lists and trees, and similar mechanisms for inductively defined types and functions.

5.1.2.4 Convenient Syntax

There are basically two aspects to the question of syntactic convenience: familiarity and ease of expression, and utility for documentation and review. The latter is somewhat less important if the language is used in an environment that includes typesetting for documentation. The former hinges on whether the language accommodates the user – for example, providing infix operators for standard arithmetic operations and familiar forms of function application including the use of delimiters and punctuation – or whether the user must accommodate the language, adjusting, for example, to Lisp-style prefix notation.

5.1.2.5 Diagrammatic Notation

Diagrammatic notation, including graphic notations as found, for example, in Statemate [H^+90], and tabular notations, as found in Parnas's "four variable method" [vSPM93], SCR [HJL95], and RSML [LHHR94], provide a specification format that can be readily understood and easily communicated. These notations typically support an underlying methodology for specification and refinement. The challenge is to provide the benefits of a diagrammatic style with sufficient underlying formality to support a range of formal analysis techniques.

5.1.2.6 Strong Typing

Strong typing is often considered a significant asset in specification languages as well as in programming languages. The difference is that specification languages can have much richer type systems than programming languages because the types do not have to be directly implementable.[9] The benefits of strong typing include economy and clarity of expression, a discipline that encourages precision, and an effective basis for mechanized typechecking. A typechecker is a program that checks that the type discipline is maintained throughout the specification; entities must match their declarations and be combined only with other entities of the same (or a compatible) type. Predictably, the actual utility of the typechecker (for detecting faults, inconsistencies, and omissions) depends both on the logical foundation underlying the specification language and on the diligence and skill of its implementors. For example, it is difficult to provide strict typechecking for languages based on set theory without sacrificing some of the flexibility of these languages because set theory doesn't provide an intrinsic notion of type. On the other hand, type theory (higher-order logic) is an inherently typed system, and languages based on higher-order logic readily support strict typechecking.

---

[8] Updating or constructing new values of structured types from existing values in a purely functional way (analogous to assignment to array elements or record fields in imperative programming languages) is also referred to as *overriding*.

[9] There are exceptions, such as the abstract or virtual class constructs in C + +, but the generalization is nevertheless a useful one.

Nevertheless, there are certain caveats. Lamport has argued against the unquestioned use of typed formalisms, noting that types are not harmless – they potentially compromise the simplicity and elegance of mathematics and complicate formal systems for mathematical reasoning [Lam95]. Strongly typed languages that do not provide overloading and type inference can be notationally complex and frustrating to use. For example, in many specification languages, addition on integers is often a different function from addition on the reals, but by "overloading" the symbol + and exploiting context to "infer" the correct addition function, the burden of the complexity falls on the system rather than on the user. The sophistication of type inference mechanisms varies; systems based on higher-order logic that provide rich type and modularization facilities require particularly sophisticated type inference mechanisms for effective user support.

If a rich type system is supported by mechanized typechecking integrated with theorem proving so that typechecking has access to theorem proving, the expressiveness of the language can be further enhanced. For example, much of the expressive power of the PVS language is achieved through the use of predicate subtypes where a predicate is used to induce a subtype on a parent type. However, the introduction of subtypes makes typechecking undecidable, requiring the typechecker to generate proof obligations (known as Type-Correctness Conditions (TCCs)) that must be discharged before the specification can be considered type correct.[10]

5.1.2.7 Total versus Partial Functions

A total function maps every element of its domain to some element in its range, whereas a partial function maps only some elements of its domain to elements of its range, leaving others undefined. While most traditional logics incorporate the assumption that functions are total, partial functions occur naturally in the kinds of applications undertaken with formal methods. Given that most logics assume that functions are total, providing a logical basis for a specification language that admits partial functions tends to be problematic. Although some recent logics (including those of VDM [Jon90], RAISE [Gro92], three-valued logics [Urq86,RT52,Res69,KTB88], and Beeson's logic of partial terms [Bee86]) allow partial functions, they typically formalize partial functions at the expense of complicating theorem proving for all specifications, even those that do not involve partiality. On the other hand, treating all functions as total in languages with only elementary type systems also has undesirable consequences, in particular, the awkwardness of having to specify normally undefined values (for example, having to specify division by zero). Total functions are less problematic in languages that support subtypes and dependent types, as illustrated previously by the PVS specification of division on the rationals as a total operation on the domain consisting of arbitrary numerators and nonzero denominators, where the latter was defined by the predicate subtype, *nonzero_rational*.

---

[10] The standard PVS example is that of the division operation (on the rationals), which is specified by `/:[rational, nonzero_rational → rational]` where `nonzero_rational:type = {x:rational | x ≠ 0}` specifies the nonzero rational numbers. The definition of division constrains all uses of the operation to have nonzero divisors. Accordingly, typechecking a formula such as `x ≠ y ⊃ (y-x)/(x-y) < 0` generates the TCC `x ≠ y ⊃ (x-y) ≠ 0` to ensure that the occurrence of the division operation is well-typed. Note the use of the "context" `(x ≠ y)` as an antecedent in the TCC. Most (true) TCCs generated in the PVS system are quickly and automatically discharged by the prover during typechecking without user intervention.

5.1.2.8 Refinement

Specification languages that support refinement provide an explicit formal basis for the hierarchical mappings used to verify successive steps in the development from abstract requirements and high-level specification to code. Although most specification languages allow refinement to be expressed, if somewhat painfully, explicit support for refinement confers a distinct advantage for describing the systematic and provably correct "implementation" of a higher-lever specification by a lower-level one.[11]

5.1.2.9 Introduction of Axioms and Definitions

In the introduction to this chapter, it was noted that a specification typically consists of a collection of axioms and definitions. Axioms can assert arbitrary properties over arbitrary (new or existing) entities. Definitions are axioms that are restricted to defining new concepts in terms of known ones. This difference has important implications; axioms can introduce inconsistencies, whereas well-formed definitions cannot. Specification languages differ with respect to facilities for introducing axioms and definitions, including the rigor with which they guarantee that axioms are consistent and definitions well-defined. Some specification languages do not allow the introduction of axioms. Although this avoids the problem of inconsistency, it can create others. For example, axioms are particularly useful for stating assumptions about the environment and the inability to define such constraints axiomatically can present a considerable drawback. On the other hand, the ability to introduce axioms should always be offset by a method (and, ideally, mechanical support) to demonstrate their consistency. While some languages prohibit arbitrary axiomatizations, others offer little or no assurance that definitions are well-formed, that is, constructed according to a *definitional principle* appropriate to the given (specification) language. The role of this principle is to ensure what is referred to as a *conservative extension* to a theory.

> "A theory A is an 'extension' of a theory B if its language includes that of B and every theorem of B is also a theorem of A; A is a 'conservative'

---

[11] Refinement is a topic that is not covered in this volume. A representative sample of the work in this area, including both model-based and algebraic approaches, may be found in the proceedings of recent workshops on refinement, including [dBdRR89, MW90], as well as in [BHL90].

> extension of B if, in addition, every theorem of A that is in the language of B is also a theorem of B [Rus93b, p. 58]."

The richness of the underlying logic, the strength of the definitional principle, and the degree and power of the associated mechanization determine the nature and extent of the concepts that may be defined in a language. Recursive definitions are an example. The problem with recursive definitions is that they may not terminate on certain arguments, that is, they may be partial rather than total. There are various strategies for extending a definitional principle to recursive definitions. One strategy is to provide a fixed template for recursive definitions along with a meta-proof that establishes that all correct instantiations of the template terminate. The strategy used in PVS is to prove well-foundedness using a technique based on a "measure" function whose value decreases across recursive calls and is bounded from below.[12] The classic example, factorial, is defined in PVS as follows, where the `MEASURE` clause specifies a function to be used in the termination proof. In this case, the measure is simply the (generic) identity function supplied by the PVS prelude.

```
factorial(x:nat): RECURSIVE nat =
IF x = 0 THEN 1 else x * factorial(x-1) ENDIF
MEASURE id
```

This definition generates a type well-formedness condition that must be discharged before the definition is considered type correct. The condition states that for all natural numbers, `x`, either `x = 0` or `x - 1` is strictly less than `x`.

Another type of definitional principle, called a "shell", provides a compact way to specify new structured types in terms of constructors, recognizers, and accessors that respectively construct new elements of the type, recognize bona fide (sub)elements of the type, and access (sub)elements.[13] This concise specification is expanded schematically to generate the axioms necessary to establish the consistency of the definition, and (in some cases) to provide other useful constructs such as induction schemes. The consistency of the axioms is assured by a meta-proof on the shell principle. Boyer and Moore make extensive use of the shell principle, axiomatizing fundamental objects including the natural numbers, literal atoms, and ordered pairs, as well as new types. PVS uses a similar, but somewhat more sophisticated shell mechanism to define abstract data types [Sha93]. The ubiquitous example of a pushdown stack can be very concisely specified in PVS.

```
 stack[ t: TYPE]: DATATYPE
   BEGIN
     empty: emptystack?
     push(top: t, pop: stack): nonempty_stack?
   END stack
```

---

[12] The template approach is more restrictive, but easier to implement; it does not require theorem proving to establish the well-definedness of a definition as does the measure function strategy.

[13] The name "shell" was first introduced by Boyer and Moore [BM79, pp. 35-40], who note that their shells were inspired by Burstall's "structures" [Bur69].

`empty` and `push` are the constructors, `empty_stack?` and `nonempty_stack?` are the recognizers for the corresponding constructors, and `top` and `pop` are the accessors for nonempty stacks. When `stack` is typechecked, a new PVS theory, `stack_adt`, is generated that consists of approximately a page and a half of PVS and provides the axioms and induction principles to ensure that the datatype is well-formed.

The distinction between definitional versus axiomatic specification is revisited in Section 5.2, where the implications of the two styles are discussed. The point of this somewhat long excursion has been to underscore the utility of both approaches; powerful definitional principles and arbitrary axiomatizations each have a role in formal specification, and a specification language that provides both accompanied by suitable mechanization is a potentially more productive tool than a language that effectively supports one approach to the exclusion of the other.

5.1.2.10 Encapsulation Mechanism

Mechanisms that provide the ability to modularize and encapsulate are as important in specification languages as they are in programming languages. Mechanisms that not only support modularization, but also allow parameterization of the modules provide even greater utility because they encourage reuse. For example, a sorting module can be defined generically and parameterized by the type of entity to be sorted and the ordering to be used, thereby allowing a single module to be (re)used to sort entities of different types according to different ordering relations. In PVS, such a module (called a `THEORY`) might appear as follows, where the idea is to sort sequences of type `T` with respect to the ordering relation `<=`. The signature of this relation indicates that `<=` takes two elements of type `T` and returns a Boolean value.

```
  sort[T:TYPE, <=: [T,T -> bool]] : THEORY
```

To ensure that instantiations are appropriate, for example, that the values provided to the ordering relation in fact constitute an appropriate ordering, semantic constraints are associated with the instantiations. There are various mechanisms for accomplishing this, including attaching assumptions to the formal parameters of the module, as in PVS. For example, it may be useful to constrain `<=` to be a preorder (that is, reflexive and transitive).[14]

```
sort[T:TYPE, <=: [T,T -> bool]: THEORY
BEGIN
ASSUMING
  pre_order: ASSUMPTION pre_order?(<=)
ENDASSUMING
END sort
```

This assumption must be discharged whenever the module `sort` is instantiated.

---

[14] `pre_order?` is a predicate defined in the PVS prelude [OSR93a].

5.1.2.11 Built-in Model of Computation

Most applications of formal methods involve reasoning about computational processes. In the discussion of discrete domain models (Section 4.3), it was noted that some specification languages have a built-in model of computation, for example, in the form of a process algebra as in LOTOS [ISO88] or certain programming-language constructions, such as the concurrency mechanisms offered in Gypsy [GAS89]. If a model of computation is present in a language, it is important to ensure that the computational model is suitable for the application at hand. For example, a study of synchronization algorithms cannot very well be performed in a notation based on synchronous communication [Rus93b, p. 162]. On the other hand, many specification languages do not incorporate a model of computation, or incorporate only a very elementary model, such as functional composition. Using functional application and composition, almost any logic can represent sequential computation. Languages such as PVS that are based on classical higher-order logic are typically rich enough to specify more complex computations, such as those involving imperative, concurrent, distributed, and real-time algorithms. For example, important properties of distributed systems can often be described and analyzed using recursive functions [LR93b].

5.1.2.12 Executability

Executability provides a pragmatic approach to exploring and debugging specifications, and to developing and evaluating test cases. Further discussion of executability may be found in Sections 5.4 and 6.3.

5.1.2.13 Maturity

The advantages of a mature specification language are similar to those of a mature programming language: documentation is reasonably accessible and complete, tool support is available and generally reliable, there is a reasonably large body of associated literature and applications, and there is some measure of standardization so that a specification written in the language provides an unambiguous and generally accepted description.

5.2 Formal Specification Styles

Specification style has various implications, ranging from readability to ease of proof. As Srivas and Miller note in reference to the formal verification of a commercial microprocessor (arguably the most ambitious microprogram verification undertaken to date) [SM95a, p. 31]: “One of the more important lessons learned during this project was to more carefully consider the trade-offs between ... styles of specification.” Srivas and Miller are specifically referring to a constructive versus a descriptive style of specification, also known as *model-oriented* versus *property-oriented*, respectively.[15] A constructive or model-oriented style is typically associated with the use of definitions, whereas a descriptive or property-oriented style is generally associated with the use of axioms. For example, consider the `mod` function, which returns the remainder when one natural number is divided by another. `mod` can be specified constructively by defining a recursive function that returns the appropriate value, or descriptively by axiomatizing certain of its number theoretic properties [SM95a, p. 28]. The descriptive style encourages *underspecification*—specifying less rather than more, and doing so as abstractly as possible—thereby avoiding the tendency to focus on *how* a concept is realized rather than simply *what* is required of it, whereas the constructive style tends to promote *overspecification*—specifying more rather than less and doing so in greater detail and specificity than necessary—thereby allowing an implementation bias to creep in earlier than warranted. On the other hand, descriptive or axiomatic specifications can introduce inconsistencies and can be less easily read and understood by the uninitiated reader than constructive specifications. Constructive specifications also tend to correspond more naturally to the procedural requirements used in many applications. Ultimately, the trade-offs between the two styles must be arbitrated by the application and by the options provided by the specification language used. Again, Srivas and Miller's experience is instructive.

> "It became evident that [the descriptive style was] in many ways a preferable style of specification . . . more readable, simpler to validate, and . . . closer to what a user wanted to know. . . . Using this style would have made specifying the core set of 13 instructions much simpler. However, doing so also would have made it easier to introduce inconsistencies in the specification. . . . The declarative [*sic*, that is, descriptive] style of specification is better-suited for reasoning with complex instruction sets [SM95a, pp. 30-31]."

Many applications can benefit by the judicious use of both styles. One approach is to use a property-oriented axiomatization as a top-level specification and introduce a suitable number of specification layers between the property-oriented requirements statements and increasingly detailed, (provably consistent) model-oriented descriptions, possibly culminating in an implementation-level specification. The idea is to establish that the implementation satisfies the requirements. Few analyses elaborate multiple layers—the example documented in [BHMY89,Bev89] and summarized in Section 5.3 is a notable exception; for most applications, more cost-effective strategies focus on key properties early in the life cycle.

There are other considerations that may be viewed as stylistic, including the trade-offs between a functional style of specification versus one in which the notion of state is explicitly represented, for example, using "Hoare sentences" to express pre- and post-conditions on a state.[16] Some specification languages support both styles, while others support only an implicit notion of state. If the notion of state is implicit, the model of computation may be more or less explicit. For example, if the specification of a control system must support the analysis of properties characterizing the evolution of the system over time, the (monitored, controlled, and state) variables are typically represented as *traces*, that is, functions from "time" to the type of value concerned, where time represents a frame, cycle, or iteration count. Purely functional specifications are intrinsically closer to ordinary logic and therefore tend to support more effective theorem proving than specifications that involve state. In general, specifications involving state tend to be unnecessarily constructive for earlier life cycle applications; functional style specifications are often adequate for the requirements and high-level design phases.

---

[15] Other terminology is also found in the literature; for example, the term "prescriptive" is sometimes used to refer to a constructive style of specification and "declarative" to a descriptive style.

5.3 Formal Specification and Life Cycle

One approach to integrating formal specification with system development is to construct a hierarchy of specifications at different levels of abstraction, each level corresponding to a different phase of the software life cycle and each level elaborating or "refining" the immediately preceding level. Using formal proof to establish that each level of the design is a correct implementation of its immediate ancestor, it is possible to develop a proof chain that automatically demonstrates that required properties are satisfied at all levels – from the requirements specification down through the implementation (code level). Such proofs typically use a mapping function that relates the objects of one level with the objects of the immediately preceding level and prove that the mapping is preserved through all possible executions. Needless to say, hierarchical specification over multiple levels is an arduous and costly undertaking. The "CLI short stack" [Bev89] – a mechanical verification of a multilevel system from an applications program in a high-level language down through the gate-level design of a microprocessor with intermediate levels including a compiler, assembler, and linker – exemplifies this approach. The LaRC verification of a reliable computing platform for real-time control is another of the few extant examples [BDH94].

Formal specification is typically most cost-effective early in the life cycle of a system. This is true for several reasons, notably the effectiveness of conventional verification and validation activities later in the life cycle versus earlier, when there is an acknowledged dearth of effective strategies and tools, and the difficulty of formal specification during the later life cycle, in the context of highly detailed, implementation-specific models. This rationale dovetails nicely with the largely pragmatic considerations that have focused most applications of formal methods on critical or key properties rather than on "total correctness." As a result, formal specification is most productively used as an integral part of the iterative development of requirements and high-level design, rather than as a one-time, benedictory activity at the end of the process.

---

[16] As an antidote to then-current program verification approaches that generated verification conditions (VCs) from programs annotated with logical assertions, yielding VCs that were difficult to map back to the original program (and the user's intuition), Hoare extended the logic to include program elements, thereby allowing the user to reason directly about programs [Hoa69].

5.4 The Detection of Errors in Formal Specification

There are several potential sources of error in a formal specification:

- It can say too little or underspecify, that is, be incomplete
- It can say too much or overspecify, that is, be overly prescriptive, thereby unnecessarily constraining later phases of the life cycle
- Or, it can be wrong, that is, it can be internally inconsistent or it can specify something anomalous or unintended.

Overspecification is difficult to detect mechanically and typically requires considerable experience to recognize and avoid.[17] The other faults are generally more amenable to the types of fault detection discussed below. Including formal proof, there are basically five regimens for detecting anomalies in a specification. The last four of these can be effectively mechanized and typically occur in the order given, since there is no point in attempting proofs on a specification that is not syntactically and semantically correct. By the same token, there is no point in checking for semantic anomalies in a specification that is not syntactically well-formed. On the other hand, each of these techniques has a particular utility, and an integrated approach that exploits the strength of each is undeniably the most effective. In some cases, this integration is inherent in a system, for example, cooperating decision procedures in a theorem prover, or the tight coupling of a typechecker and a proof checker to provide strict typechecking in the presence of non trivially decidable properties. In other cases, the integration is achieved by judicious use of available techniques, for example, "prototyping" a potentially difficult and costly proof by using model checking, simulation, or animation to examine a finite case before attempting the more general proof with a theorem prover or proof checker. In any case, the utility of the fault-detection techniques discussed below can be significantly enhanced by exploiting the potential synergy created by their judicious combination.

Inspection: Inspections run the gamut from informal peer review to well-defined, formalized procedures. The Fagan-style inspections discussed in Section 3.2 are among the most frequently used quasi-formal inspections. In theory, these manual inspections can detect all the error types noted above, although in practice, manual inspections are not as effective as mechanized tools in detecting subtle or deep-seated anomalies, such as logical inconsistencies and (unintended) implications, or in consistently locating semantic or even syntactic errors in specifications. Nevertheless, Fagan-style inspections and other similarly exacting inspection methods can effectively complement formal methods, and vice versa. The AAMP5 microprocessor project illustrates this point nicely. Miller and Srivas note the surprising

---

[17] Jones characterizes a notion of overspecification or implementation bias for constructively defined specifications. Briefly, a specification is biased with respect to a given set of operations "if there exist different elements of the [underlying] set of states which cannot be distinguished by any sequence of the operations." [Jon90, pp. 216-219].

> "extent to which formal specifications and inspections complemented each other. The inspections were improved by the use of a formal notation, reducing the amount of debate over whether an issue really was a defect or a personal preference. In turn, the inspections served as a useful vehicle for education and arriving at consensus on the most effective styles of specification. This is reflected in ... the lower number of defects recorded in the later inspections [MS95, p. 9]."

As this quote suggests, the symbiotic relationship between formal methods and conventional inspection techniques provides a natural medium for technology transfer.

Parsing: Parsing is a form of analysis that detects syntactic inconsistencies and anomalies, such as misspelled keywords, missing delimiters, or unbalanced brackets or parentheses. Parsing guarantees (only) that a specification conforms to the syntactic rules of the formal specification language in which it is written.

Typechecking: Typechecking is a form of analysis that detects semantic inconsistencies and anomalies, such as undeclared names or ambiguous types. As noted in Section 5.1.2, formal specification languages based on higher-order logic admit effective typechecking, while in general, those based on set theory do not. When available, strict typechecking is an extremely effective way of determining whether a specification makes semantic sense. Again as noted in Section 5.1.2, the type system of a specification language may not be trivially decidable, in which case typechecking is similarly undecidable and proof obligations must be generated and discharged before the specification is considered typechecked.

Execution (Simulation/Animation): Direct execution, simulation, and animation offer further options for detecting errors in a specification. If a formal specification language is directly executable, or contains a directly executable subset, execution and animation can be accommodated in the same formally rigorous context in which the specification is developed. If not, the formal specification must be reinterpreted into high-level, dynamically executable program text that bears no formal relation to the original specification (see [MW95, Chapter 5] for an example of the latter). Some languages offer both, that is, a directly executable subset, as well as the option of user- or system-defined program text to drive animation of nonexecutable parts of the specification. The concrete representation of algorithms and data structures required by most finite-state enumeration and model-checking methods (see below) makes them directly comparable to direct execution techniques, as found, for example, in the VDM-SL Toolbox [VDM]. In some cases, model checkers also provide simulation. For example, the reachability analysis strategy used by state-exploration model checkers can also be used to "simulate" system behavior by exploring a single path (rather than all possible paths) through the state space. Both Murϕ [DDHY92, ID93] and SPIN [Hol91] can simulate the execution of models written in their respective languages. The type of errors found by direct execution techniques varies, depending on other error detection techniques, if any, used prior to simulation or animation. For example, [MW95, p. 92] animated a specification that had previously undergone only syntactic analysis and weak type analysis (essentially limited to arity checking on function and operation calls). In their case, animation detected two type errors in addition to errors due to misinterpretation of the requirements, incorrect specification of requirements, and erroneous translation from the specification into the simulation language. Executability also supports the development and systematic evaluation of test suites, thereby potentially exposing flaws and oversights in a test regime, as well as in the corresponding specification.

Theorem Proving, Proof Checking, and Model Checking: Theorem proving, proof checking, and model checking are all forms of analysis that can be used to detect logical anomalies and subtle infelicities in a formal specification. Although historically these forms of validation were used to prove correctness of programs and detailed hardware designs, they are now typically used for fault detection and design exploration, where they are arguably most effective, as well as for verifying correctness. The analysis provided by theorem proving, proof checking, and model checking not only involves the specification, but also its logical consequences, that is, all formulas that can be proved from the original specification using formal deduction. There are several issues in the validation of formal specifications. One is the issue of internal consistency, that is, whether the specification is logically consistent. If not, the specification fails to say anything useful. Another is the issue of meaningfulness, that is, whether the specification means what is intended. A third is the issue of completeness. Although various notions of completeness have been proposed, the general idea is that a specification should identify all contingencies and define behavior appropriate to each.[18] The type of testing and error detection offered by theorem proving, proof checking, and modeling is in many ways analogous to traditional testing regimes; the theorem prover, proof checker, or model checker "executes" the specification, allowing the practitioner to explore design options and the implications of design choices.

5.5 The Utility of Formal Specification

A specification may serve many different functions. Lamport [Lam89, p. 32] has suggested that a formal specification functions as "a contract between the user of a system and its implementer. The contract should tell the user everything he must know to use the system, and it should tell the implementer everything he must know about the system to implement it. In principle, once this contract has been agreed upon, the user and the implementer have no need for further communication." Lamport's simile highlights three issues. First, as noted earlier, one of the most important functions of a formal specification is analytic; using the deductive apparatus of the underlying formal system, a formal specification serves as the basis for calculating, predicting, and (in the case of executable specifications) testing system behavior. However, a formal specification may also serve an important descriptive function, that is, provide a basis for documenting, communicating, or prototyping the behavior and properties of a system. Second, a (completed) specification represents the formalization of a consensus about the behavior and properties of a system. Diverging somewhat from Lamport's description and focusing on the early life cycle, we prefer to view a formal specification as a contract between a client, a requirements analyst (and possibly also a designer), and a formal methods practitioner. Third, while in principle, a finalized contract precludes the need for further communication among the interested parties, in practice, moving from informal requirements to a formal specification and high-level design is an iterative rather than a linear process; issues exposed in the development of the formal specification may need to be factored back into the requirements, and similarly, issues raised by the high-level design may percolate back to impact either the formal specification, the requirements, or both.

---

[18] Rushby [Rus93b, pp. 69-71] cites several specialized definitions, including characterizations of completeness for abstract data types and for real-time process-control systems.

Although a specification that has not been validated through proof can be aptly compared to a program that has not been debugged, there are nevertheless real benefits to be gained from modeling and formally specifying requirements and high-level designs, including the following.

- *Clarify Requirements and High-Level Designs:* A formal specification provides a concise and unambiguous statement of the underlying requirements and design, thereby exposing fundamental issues that tend to be obscured by lengthy informal statements. The formalization of the requirements for the recent optimization of Space Shuttle Reaction Control System Jet Selection (JS) [NASA93, Appendix B] recounted here in Section 3.1.1 illustrates this point nicely.

- *Articulate Implicit Assumptions:* Formalisms can help identify and express implicit assumptions in requirements and design. For example, the concept of state variables is not explicitly mentioned in Space Shuttle requirements; their existence must be inferred from context by noting the function and persistence of local variables. Explicitly modeling and specifying state variables can significantly increase the precision and perspicuity of the requirements, as illustrated by the partial specification of the new Space Shuttle Global Positioning System (GPS) navigation capability [DR96]. Identifying undocumented assumptions is particularly important in the context of an evolving system design.

  Another aspect of requirements and high-level design that frequently contains implicit assumptions is the interaction of the system with the environment or context in which it is assumed to operate, including the input space. Making input constraints and environmental assumptions explicit often exposes requirements and design-level issues that have been overlooked.[19] The specification of the Space

---

[19] The A-7 Methodology [vS90], among others, has paid particular attention to the explicit enumeration of relevant environmental variables.

Shuttle Heading Alignment Cylinder Initiation Predictor and Position Error Displays Change Request (HAC CR) is a good example of the value of the process of formalization for identifying and capturing undocumented, domain-specific assumptions. Quoting from the report for the HAC CR formal methods project: "Capturing such [domain-specific] knowledge and documenting it as rationale with the specification is valuable [RB96, p. 17]."

- *Expose Flaws:* The process of formalization invariably exposes significant flaws in requirements and high-level design, even without the benefit of analysis or proof. In the case of strongly typed specification languages, typechecking can provide a potent tool for revealing anomalies in the specification, as well as potential anomalies in the requirements and design, and doing so early in the life cycle while errors are far less costly to correct. The previously mentioned GPS project [DR96] provides a nice example of the utility of specification for revealing anomalies in immature requirements for large, complex systems, especially among subsystem interfaces. Executing a specification provides another productive means of exposing flaws, as noted in [MW95].
- *Identify Exceptions:* The discipline involved in formalizing requirements and high-level design also serves to identify "end cases" and exceptions and to encourage more thorough consideration of these exceptional cases, as illustrated in [LFB96].
- *Evaluate Test Coverage:* An executable specification may also be used to run and evaluate proposed test suites, yielding a measure of test coverage relatively early in the life cycle.

The utility of formal specification also extends to work in program transformation and synthesis, that is, the mechanical application of a series of transformations that derives a program from its specification. This approach differs from traditional compilation of high-level languages insofar as it seeks to bridge a far larger language gap between input (specification) and output (program). To make this feasible, the scope of the specification language must be severely constrained, and/or the transformation process must be guided by a skilled programmer. The techniques rely on a set of correctness-preserving transformations that guarantee that the resulting program will exhibit the same behavior as its specification. Ideally, the transformation also confers additional (desired) properties such as efficiency. Suggestive, but by no means exhaustive, examples of this broad spectrum of techniques are the following:

- Problems expressed in a specification-oriented language (for example, pure Lisp) typically exhibit clarity and simplicity, but lack the efficiency and portability that comes from a conventional programming language (for example, FORTRAN and C). Boyle has pursued a transformational approach to bridging this gap [Boy89] that involves successive decomposition into a series of steps that can be accomplished by the automatic application of a set of special-purpose, but straightforward transformations. Examples include the use of a succinct functional specification to derive a FORTRAN implementation of an algorithm for solving one-dimensional hyperbolic partial differential equations [BH91].

- A certain class of problems can be solved by a carefully programmed instance of a general algorithmic technique, for example, search problems can be solved by a divide-and-conquer strategy. KIDS (Kestrel Interactive Development System) [Smi90] provides tools for deductive inferencing, algorithm design, expression simplification, finite differencing, partial evaluation, data type refinement and other general transformations that allow a user " to synthesize complex codes embodying algorithms, data structures, and code-optimization techniques that might be too difficult to produce manually [SG96, p.31]." The approach is interactive; the user guides the system in the application of powerful correctness-preserving transformations. KIDS has been applied to a variety of domains, including scheduling, combinatorial design, sorting and searching, computational geometry, pattern matching, and mathematical programming.

- The class of finite functions, including for example, finite state transitions, lends itself to tabular representations that can be manipulated to perform various consistency and completeness checks and, in some cases, to generate code and documentation. For example, the decision table, "a tabular format for specifying a complex set of rules that choose among alternative actions" [HC95, p. 97] provides the basis for the Tablewise tool [HC95, HGH96] that tests these tables for consistency and completeness, performs a limited form of structural analysis, and generates Ada code implementing the table, as well as English-language documentation.

- If an application domain is suitably restricted, it is possible to develop a completely automatic process for synthesizing a program from its specification. The AMPHION system [LPPU94] illustrates this approach for the domain of solar system kinematics. The user specifies a problem via a graphical user interface portraying the domain's astronomical objects and desired configuration. The system then selects components from a preexisting FORTRAN subroutine library and synthesizes the "glue" code that assembles these components into a complete solution. The system applies constructive theorem proving to perform its selection and synthesis. The end user, however, operates purely at the specification level and need never interact with this underlying mechanism.

5.6 A Partial SAFER Specification

The PVS specification of SAFER is constructive in style and retains the explicit notion of state represented in the SAFER models developed at the end of Chapter 4. To facilitate readability and emphasize the mapping between informal description, requirements, and the PVS formalization, the specification also preserves the bias toward representative rather than abstract formalization introduced into the models of the preceding chapter. The complete PVS specification is presented in full in Section C.3.3. The fragment discussed below continues the focus on thruster selection. This discussion is intended to be self-contained; if additional information on the relatively few PVS language features necessary to understand the formal specification can be found in Section C.3.1. Full PVS documentation is available in [OSR93b].

The PVS specification of thruster selection is a straightforward elaboration of the underlying functional model developed in Chapter 4. Accordingly, the skeleton of the PVS theory for thruster selection shown below consists of three functions: `integrated_commands`, which forms an integrated, six degree-of-freedom command from the HCM and AAH inputs; `selected_thrusters`, which takes an integrated command and selects the thrusters necessary to achieve the command; and `selected_actuators`, which acts as an interface function and consists of the composition of `integrated_commands` and `selected_thrusters`. Each of these functions is parameterized by from one to three parameters denoted by a parameter name followed by a type name. The type definitions for these types are not reproduced here, but are available in subsequent discussion and in Appendix C either in the theory `avionics_types` or in the theory most closely associated with the object in question. For example, the types `six_dof_command` and `rot_command` are defined in the theory `avionics_types`, while the type `AAH_state` is defined in the theory `automatic_attitude_hold`. The type `actuator_commands` is defined as a `thruster_list`. Thruster selection is formalized as a PVS *theory* aptly named `thruster_selection`. The theory is the basic organizational concept in PVS and provides the modularization and encapsulation familiar in modern programming languages; theories may export to and import from other theories.

```
thruster_selection: THEORY
BEGIN

integrated_commands((HCM:   six_dof_command),
                    (AAH:   rot_command),
                    (state: AAH_state)): six_dof_command = ...

selected_thrusters(cmd: six_dof_command): thruster_list = ...

selected_actuators((HCM:   six_dof_command),
                   (AAH:   rot_command),
                   (state: AAH_state)): actuator_commands =
        selected_thrusters(integrated_commands(HCM, AAH, state))

END thruster_selection
```

Fleshing out the skeleton of `thruster_selection` introduces a type definition (`thruster_list`) and five additional functions. However, the first thing to notice about this elaborated version is the `IMPORTING` clause, which allows visible entities introduced in the theories `avionics_types`, `propulsion_module`, and `automatic_attitude_hold` to be imported and used in the theory `thruster_selection`.

```
IMPORTING avionics_types, propulsion_module, automatic_attitude_hold
```

For example, this importing clause brings in several type declarations, including those mentioned above for `six_dof_command` and `rot_command`. The importing clause is followed by a local declaration of the type `thruster_list`, which is defined as a list of `thruster_names`.[20] The type `thruster_names` is in turn imported from the theory `propulsion_module`.

```
thruster_list:      TYPE = list[thruster_name]
```

The next declaration introduces the Boolean-valued function `rot_cmds_present`, whose signature includes one parameter of type `rot_command`.

```
rot_cmds_present(cmd: rot_command): bool =
       (EXISTS (a: rot_axis): cmd(a) /= ZERO)
```

The declaration for `rot_command` defines a function from type `rot_axis` to type `axis_command`.

```
rot_axis:       TYPE = {roll, pitch, yaw}
axis_command:   TYPE = {NEG, ZERO, POS}
rot_command:    TYPE = [rot_axis  -> axis_command]
```

`rot_axis` is an enumerated type corresponding to the three rotation axes: roll, pitch, yaw. `axis_command` is an enumerated type with three values corresponding to the HCM or AAH command values: negative, zero, or positive. The notation `cmd(a)` denotes a function that maps a rotation axis (one of: roll, pitch, yaw) to the command associated with that axis (one of: NEG, ZERO, POS). `rot_cmds_present` returns the value of the existentially quantified formula shown above. That value is true if there is at least one rotational axis whose associated (HCM or AAH) command is nonzero, and false otherwise.

The next function, `prioritized_tran_cmd`, specifies the requirement that there is at most one translation command at a given cycle and that translation axis commands are prioritized with `X`-axis commands having highest priority and `Z`-axis commands lowest priority. The encoding takes the form of a nested-if expression and uses a PVS *override* expression to derive a new value from `null_tran_command`, which is written as an unnamed function or *lambda* expression. The result of an override expression is a function[21] that is exactly the same as the original, except that it takes on new values at the specified arguments. A `tran_command` does the analogous mapping for the translation axes, `X`, `Y`, and `Z` that the `rot_command` does for the rotation axes. Accordingly, in the first branch of the nested-if expression, if an `X`-axis command is present (the value of `tran(X)` is not equal to `ZERO`), `null_tran_command` takes on the value of `tran(X)` for the argument `X`, and similarly for the other branches of the nested-if, which handle the cases for `Y`- and `Z`-axis updates.

---

[20] The `thruster_list` declaration actually uses the built-in `list` datatype provided in the PVS prelude [OSR93a, pp. 39–41,78–80], [Sha93].

```
tran_axis:      TYPE = {X, Y, Z}
tran_command:   TYPE = [tran_axis -> axis_command]
null_tran_command: tran_command = (LAMBDA (a: tran_axis): ZERO)

prioritized_tran_cmd(tran: tran_command): tran_command =
        IF tran(X) /= ZERO
           THEN null_tran_command WITH [X := tran(X)]
        ELSIF tran(Y) /= ZERO
           THEN null_tran_command WITH [Y := tran(Y)]
        ELSIF tran(Z) /= ZERO
           THEN null_tran_command WITH [Z := tran(Z)]
           ELSE null_tran_command
        ENDIF
```

The function `combined_rot_cmds` transforms rotation commands from the HCM and the AAH and returns a “combined” rotation command that inhibits HCM commands at the time AAH is initiated (`ignore_HCM`), but otherwise gives nonzero HCM rotation commands precedence over AAH rotation commands. The argument `ignore_HCM` is a predicate, that is, a function with range type Boolean. Note the use of the lambda expression to map over the three rotation axes.

```
rot_predicate:     TYPE = [rot_axis -> bool]

combined_rot_cmds((HCM_rot:    rot_command),
                  (AAH:        rot_command),
                  (ignore_HCM: rot_predicate)): rot_command =
        (LAMBDA (a: rot_axis):
                IF HCM_rot(a) = ZERO OR ignore_HCM(a)
                   THEN AAH(a)
                   ELSE HCM_rot(a)
                ENDIF)
```

Using the preceding definitions, `integrated_commands` is elaborated as shown below. The only new bit of PVS that requires explanation is the record structure used to specify the integrated six degree-of-freedom command. In PVS, record types take the form

---

[21] Or record; a PVS record may also be modified by an override expression.

$$[\# \; a_1 : t_1, \ldots a_n : t_n \; \#]$$

where the a_i are the accessors and the t_i are the component types. Record access in PVS uses functions and functional notation, for example, $a_i(r)$, rather than the more usual "dot" notation $r.a_i$. Elements of the PVS record type (or, equivalently, record constructors) have the form

$$(\# \; a_1 : t_1, \ldots a_n : t_n \; \#)$$

For example, the record type `six_dof_command` has two accessors, one each of type `tran_command` and type `rot_command`. In other words, an integrated six degree-of-freedom command has two components representing the commanded acceleration in the translational and rotational axes. Since both components are modified, record constructors rather than override expressions are used. Details of the `AAH_state` record type have been suppressed below, but appear in full in Appendix C. The requirement that HCM rotation commands suppress HCM translation commands, but HCM translation commands may coexist with AAH rotation commands, is specified by the two branches of the if-expression.

```
rot_cmds_present(cmd: rot_command): bool =
        (EXISTS (a: rot_axis): cmd(a) /= ZERO)

six_dof_command:   TYPE = [# tran: tran_command, rot: rot_command #]

AAH_state:         TYPE = [# ignore_HCM:   rot_predicate, ...     #]

integrated_commands((HCM:   six_dof_command),
                    (AAH:   rot_command),
                    (state: AAH_state)): six_dof_command =
        IF rot_cmds_present(rot(HCM))
           THEN (# tran := null_tran_command,
                   rot  := combined_rot_cmds(rot(HCM), AAH,
                                             ignore_HCM(state)) #)
           ELSE (# tran := prioritized_tran_cmd(tran(HCM)),
                   rot  := AAH #)
        ENDIF
```

Astute readers may have noticed that this version of `integrated_commands` does not take into account the additional requirement that AAH is disabled on an axis if a crewmember rotation command is issued for that axis while AAH is alive, resulting in the possibility reflected in the model in Chapter 4 as the transition "three axes off," where all three axes have been disabled in this way. Actually, the version of `integrated_commands` presented above is slightly simplified; the full version in Appendix C does handle this case.

The next two functions, `BF_thrusters` and `LRUD_thrusters`, specify the thruster select logic presented in the tables in Figures C.2 and C.3, respectively. The details are omitted here, but the full version in Appendix C specifies these tables using nested PVS tables that yield admirable traceability between the documentation and the specification.

```
BF_thrusters(X_cmd, pitch_cmd, yaw_cmd: axis_command): thruster_list = ...

LRUD_thrusters(Y_cmd, Z_cmd, roll_cmd: axis_command):  thruster_list = ...
```

The elaborated version of `selected_thrusters` reveals somewhat more about how an integrated six degree-of-freedom command is mapped into a vector of actuator commands. The specification uses a PVS *let* expression, a syntactic convenience that allows the introduction of bound variable names to refer to subexpressions. In this case, the bound variables refer to the back/front (BF) and the left/right/up/down (LRUD) thrusters defined by the thruster select logic (specified as `BF_thrusters` and `LRUD_thrusters`) to implement the commanded translational and rotational accelerations. The resulting list of thrusters is formed by appending the BF and LRUD thruster lists.

```
selected_thrusters(cmd: six_dof_command): thruster_list =
        LET BF_thr =
              BF_thrusters(tran(cmd)(X), rot(cmd)(pitch), rot(cmd)(yaw)),
            LRUD_thr =
              LRUD_thrusters(tran(cmd)(Y), tran(cmd)(Z), rot(cmd)(roll))
        IN  append(BF_thr, LRUD_thr)
```

Once again, the function presented here is a somewhat simplified version of the specification in Appendix C. In this case, the simplification has been to omit the logic corresponding to the rightmost two columns of Figures C.2 and C.3, which specify the use of two additional thrusters for certain commanded accelerations if the given constraints are satisfied. For example, the thruster select logic for "-X, -pitch, -yaw" (first row of the table in Figure C.2) specifies thruster B4 and, conditionally, thrusters B2 and B3; B2 and B3 are selected only if there is no commanded roll.

The final function in theory `thruster_selection` is the interface function `selected_actuators`, which was previously introduced as it appears in Appendix C. The somewhat abbreviated version of the full theory discussed here is collected in full below. Note that type declarations from other theories reproduced above to facilitate the discussion do not explicitly appear, but are implicitly "visible" via the `IMPORTING` clause.

```
thruster_selection: THEORY
BEGIN

IMPORTING avionics_types, propulsion_module, automatic_attitude_hold

thruster_list:      TYPE = list[thruster_name]

rot_cmds_present(cmd: rot_command): bool =
        (EXISTS (a: rot_axis): cmd(a) /= ZERO)

prioritized_tran_cmd(tran: tran_command): tran_command =
        IF tran(X) /= ZERO
           THEN null_tran_command WITH [X := tran(X)]
        ELSIF tran(Y) /= ZERO
           THEN null_tran_command WITH [Y := tran(Y)]
        ELSIF tran(Z) /= ZERO
           THEN null_tran_command WITH [Z := tran(Z)]
           ELSE null_tran_command
        ENDIF

combined_rot_cmds((HCM_rot:    rot_command),
                  (AAH:        rot_command),
                  (ignore_HCM: rot_predicate)): rot_command =
        (LAMBDA (a: rot_axis):
                IF HCM_rot(a) = ZERO OR ignore_HCM(a)
                   THEN AAH(a)
                   ELSE HCM_rot(a)
                ENDIF)

integrated_commands((HCM:   six_dof_command),
                    (AAH:   rot_command),
                    (state: AAH_state)): six_dof_command =
        IF rot_cmds_present(rot(HCM))
           THEN (# tran := null_tran_command,
                   rot  := combined_rot_cmds(rot(HCM), AAH,
                                             ignore_HCM(state)) #)
           ELSE (# tran := prioritized_tran_cmd(tran(HCM)),
                   rot  := AAH #)
        ENDIF
```

```
BF_thrusters(X_cmd, pitch_cmd, yaw_cmd: axis_command): thruster_list = ...

LRUD_thrusters(Y_cmd, Z_cmd, roll_cmd: axis_command):  thruster_list = ...


selected_thrusters(cmd: six_dof_command): thruster_list =
        LET BF_thr =
              BF_thrusters(tran(cmd)(X), rot(cmd)(pitch), rot(cmd)(yaw)),
            LRUD_thr =
              LRUD_thrusters(tran(cmd)(Y), tran(cmd)(Z), rot(cmd)(roll))
        IN  append(BF_thr, LRUD_thr)

selected_actuators((HCM:   six_dof_command),
                   (AAH:   rot_command),
                   (state: AAH_state)): thruster_list =
        selected_thrusters(integrated_commands(HCM, AAH, state))

END thruster_selection
```

Chapter 6

Formal Analysis

Formal analysis refers to a broad range of tool-based techniques that can be used singly or in combination to explore, debug, and verify formal specifications, and to predict, calculate, and refine the behavior of the systems so specified. These analysis techniques, which differ primarily in focus, method, and degree of formality, include direct execution, simulation, and animation; finite-state methods (state exploration and model checking); and theorem proving and proof checking.

This chapter describes each of these techniques and suggests strategies for their productive combination. It also examines the role of proof in theory interpretation, proofs of properties, and enhanced typechecking, as well as the utility of the proof process for calculation, prediction, and verification. The issue of mechanized support for formal analysis is presented, albeit in a suggestive rather than exhaustive discussion. The chapter closes with the specification and proof of the SAFER requirement that describes the maximum number of thrusters that can be fired simultaneously.

6.1 Automated Deduction

Automated deduction or theorem proving refers to the mechanization of deductive reasoning. Deductive methods provide a foundation for reasoning about infinite-state systems and are typically preferred for abstract, high-level specifications and data-oriented applications. There are a variety of approaches to mechanizing formal deduction, reflecting the relative maturity of the field of mechanical theorem proving. This section begins with background material on formal systems and their models, followed by a history of automated deduction, a survey of techniques underlying automated reasoning, and concluding remarks on their utility.

6.1.1 Background: Formal Systems and Their Models

The material in this section provides technical background that some readers may prefer to skip the first time through, or to detour altogether. Dangerous bend signs bracket the most technical parts of the section.

6.1.1.1 Proof Theory

A formal system consists of a nonempty set of primitives—typically a set of finite strings taken from an alphabet of symbols; a set of axioms, that is, statements, taken as given, involving the primitives; and a set of inference rules or other means of deriving further statements, called *theorems.*[1] The axioms and rules of inference of a formal system are referred to as its *deductive system.* A set of axioms, together with all the theorems derivable (provable) from it and from previously derived theorems, is called a *theory.* A proof of a theorem in a formal system is simply a series of transformations that conform to the rules of inference of that system. As such, the notion of proof is strictly syntactic. The symbol $\vdash$ (read "turnstile") is used to express this notion of proof. Thus $\vdash_L \psi$, read "ψ is provable in logic L" (or, equivalently $\vdash \psi$ if the logic is unambiguous from the context), means that ψ is a theorem in the given logic, that is, ψ is provable using the axioms of L without further assumptions. In general, the relationship between a sentence ϕ and the set of sentences, $\gamma_0, \ldots, \gamma_n$, assumed for its proof is expressed as $\gamma_0, \ldots, \gamma_n \vdash T$, where each γ_i is either an axiom, an additional assumption[2], or a previously proved theorem.

The notion of formal system sketched thus far is purely syntactic, describing what is generally referred to as an *uninterpreted calculus* or simply a calculus. The study of the purely formal or syntactic properties of an uninterpreted calculus, including deducibility, consistency, simple completeness, and independence, is called *proof theory.* The three notions of consistency, completeness, and independence are not equally important. Consistency is of fundamental importance because it provides a minimal condition of adequacy on any set of (nonintentionally self-contradictory) axioms. A formal system is *consistent* if it is not possible to derive from its axioms both a statement and the denial (negation) of that statement. The notion of completeness has many different interpretations, all of which share the idea that a formal system is complete if it is possible to derive within it all statements satisfying a given criterion. In general, completeness has theoretical importance for logicians, but less importance for those working in formal methods. It is quite difficult to establish completeness for systems of any complexity, and many interesting and even important formal systems are provably incomplete. A formal system S is said to be *simply complete* if and only if, for every closed, well-formed formula, A, either A or $\neg A$ is a theorem of S, that is, A can either be proved or disproved in S. Other terms for proof-theoretic notions of completeness include *deductively complete*, *syntactically complete*, and *complete with respect to negation.*[3] The notion of independence refers to whether any of the axioms or rules of inference of a system are superfluous, that is, can be derived from the remaining deductive system. Independence is largely a matter of "elegance." Although economy is a desirable characteristic of an axiom system, its absence does not necessarily impact the ultimate acceptibility or utility of the system.

---

[1] In this section, the terms *sentence*, *statement*, and *well-formed formula* are used interchangeably, avoiding subtle distinctions sometimes made in the literature. In the context of first-order logic, these terms are synonymous with *closed* formula and denote a formula in which there are no free (unbound) variables.

[2] Assumptions are statements assumed to be true without proof. Axioms are assumptions whose truth is assumed to be "self-evident," empirically discoverable or, in any case, stipulated for the sake of argument, rather than proved using the given rules of inference. There are *logical* and *nonlogical* axioms. The latter deal with specific aspects of a domain, for example, Peano's axioms (postulates) which are interpreted with respect to a domain of numbers, whereas logical axioms deal with general logical properties of the given calculus, for example, the axioms of propositional calculus.

A formal system, S, is *decidable* if there is an effective procedure (algorithm) for determining whether or not any closed, well-formed formula, ψ, of S is a theorem of S. Simple completeness can also be defined in terms of decidability. A formal system, S, is simply complete if it is consistent and if every closed, well-formed formula in S is decidable in S [Sho67, p. 45]. A formal system, S, is *semidecidable* if there is an algorithm for recognizing the theorems of S. If given a theorem, the algorithm must halt with a positive indication. If given a nontheorem, the algorithm need not halt, but if it does, it must give a negative indication. S is *undecidable* if it is neither decidable nor semidecidable. The propositional (statement) calculus is decidable. The predicate calculus is semidecidable, although there are subsystems of first-order predicate logic, such as *monadic predicate logic* (so-named because predicates can take only one argument), that are decidable.

In the logical tradition, the distinction between syntax and semantics largely reflects the distinction between formal systems and their interpretations, as studied by proof theory and its semantic analog, model theory, respectively. An *interpretation* consists of a (nonempty, abstract or concrete) *domain of discourse* and a mapping relative to that domain that assigns a *semantic value* or meaning to each well-formed sentence of the calculus, as well as to every well-formed constituent of such a sentence. For example, an interpretation for a predicate calculus would assign a value to function and predicate symbols, constants, and variables. The meaning or semantic value assigned to a syntactically well-formed sentence of the predicate calculus would be a truth value, either true or false, depending on the values assigned to its constituent parts. If the description of a formal system includes semantic rules that systematize an interpretation for each syntactically well-formed constituent, the calculus is said to be *interpreted.*[4]

---

[3] Gödel's proof that arithmetic is incomplete if consistent used a proof-theoretic notion of completeness.

[4] Carnap [Car58, pp. 102-3] defines a *calculus* as "a language with syntactical rules of deduction," an *interpreted language* as "a language for which a sufficient system of semantical rules is given," and an *interpreted calculus* as "a language for which both syntactical rules of deduction and semantic rules of interpretation are given."

6.1.1.2 Model Theory

An interpretation is a *model* for a formal system if all the axioms of the formal system are true in that interpretation. Similarly, an interpretation is a model for a theory or for a set of sentences if it is a model for the formal system in which the theory or the set of sentences are expressed and all the sentences in the given theory or the given set of sentences also valuate to true in that model. If a theory has an axiomatic characterization, a model for the theory is necessarily a model for the axioms of the theory. Most interesting theories have unintended (nonstandard) models, as well as intended (standard) ones. For example, plane geometry is the standard model of the Euclidean axioms, but not, as was believed before the discovery of the non-Euclidean geometries, the only model. Similarly, the natural numbers are the standard or intended model of the Peano axioms, although, again, not the only model [Kay91]. The fact that an inconsistent system cannot have a model provides both a syntactic and semantic characterization of consistency that can be usefully exploited. For example, it is typically easier to demonstrate syntactically that a system is inconsistent, deriving a contradiction from the axioms, than to use a meta-level argument to prove that the system has no models. Conversely, it is generally easier to demonstrate semantically that a system is consistent by exhibiting a model, than to show the impossibility of deriving a contradiction from the given axioms.

Model theory is the study of the interpretations of formal systems. Of particular importance are the concepts of logical consequence, validity, completeness, and soundness. Definitions of these notions reveal the rich interplay between proof theory and model theory. Let I be a set of interpretations for a calculus and ψ be a sentence of the calculus. ψ is *satisfiable* (under I) if and only if at least one interpretation of I valuates ψ to true. ψ is (universally) *valid*, written $\models \psi$, if and only if every interpretation in I valuates ψ to true.[5] If every model of a set of sentences, S, is also a model of a sentence, ψ, then S is said to *entail* ψ, written $S \models \psi$.

Let ψ be a sentence and Γ be a set of sentences $\phi_1, \ldots, \phi_n$ of a formal system, S. S is semantically complete with respect to a model M (weakly semantically complete) if all (well-formed) sentences of S that are true in M are theorems of S. A formal system, S, is *sound* if $\Gamma \models \psi$ whenever $\Gamma \vdash \psi$, that is, if the rules of inference of S preserve truth. Semantic completeness is the converse of soundness; soundness establishes that every sentence provable in S is true in S relative to M, and (semantic) completeness establishes that every sentence true in S relative to M is provable in S. Both the propositional calculus and the predicate calculus are sound and complete.

There is also a semantic characterization of independence. A given axiom, ϕ, of a formal system, S, is independent of the other axioms of S if the system, S', that results from deleting ϕ has models that are not also models of (the whole system) S. Ideally, the syntactic and semantic notions of independence are provably equivalent for a given system S. As noted with respect to the proof- and model-theoretic characterizations of consistency, a semantic argument may be easier in some cases and a syntactic one in others. However, it is apparently still an open question as to what properties a system must possess to ensure that the syntactic and semantic characterizations of independence are equivalent.

---

[5] Arguably, for the purposes of formal methods, only those interpretations that make the theorems of a formal system true, that is, only the models of the system are of interest. With this in mind, the definitions of satisfiability and validity can be stated in terms of models rather than interpretations, as done in [Rus93b, p. 223].

6.1.1.3 An Example of a First-Order Theory

Shoenfield's classical axiom system for the natural numbers, N, provides a nice illustration of a class of formal system known as a first-order theory [Sho67, pp. 22,3]. In the following definition, A, B, and C are formulas and x and y are (syntactic) variables in the given first-order language, $\mathbf{f}$ is an n-ary function symbol, and $\mathbf{p}$ is an n-ary predicate symbol. A formal system, T, is defined as

- a first-order language
- the following logical axioms, as well as certain further nonlogical axioms
  - propositional axiom: $\neg A \vee A$
  - substitution axiom: $A_x[a] \rightarrow \exists x A$
  - identity axiom: $x = x$
  - equality axiom: $x_1 = y_1 \rightarrow \ldots \rightarrow x_n = y_n \rightarrow \mathbf{f}x_1 \ldots x_n = \mathbf{f}y_1 \ldots y_n$ or $x_1 = y_1 \rightarrow \ldots \rightarrow x_n = y_n \rightarrow \mathbf{p}x_1 \ldots x_n \rightarrow \mathbf{p}y_1 \ldots y_n$
- the following rules of inference[6]
  - expansion rule: infer $B \vee A$ from A
  - contraction rule: infer A from $A \vee A$
  - associative rule: infer $(A \vee B) \vee C$ from $A \vee (B \vee C)$
  - cut rule: infer $B \vee C$ from $A \vee B$ and $\neg A \vee C$
  - $\exists$-introduction rule: if x is not free in B, infer $\exists x A \rightarrow B$ from $A \rightarrow B$

The definition of T provides the logical apparatus necessary for specifying a (first-order) theory. The only additions required are a specification of the theory's nonlogical symbols and its nonlogical axioms. For example, Shoenfield's axiomatization of the natural numbers is specified as a theory, N, with the following nonlogical symbols and axioms [Sho67, p. 22].

- nonlogical symbols: the constant 0, the unary function symbol S (denoting the successor function), the binary function symbols $+$ and $\cdot$, and the binary predicate symbol $<$.

---

[6] An occurrence of x in A is *bound in A* if it occurs in a part of A of the form $\exists x B$; otherwise, it is *free in A* [Sho67, p. 16].

- nonlogical axioms

  N1. $Sx \neq 0$

  N2. $Sx = Sy \rightarrow x = y$

  N3. $x + 0 = x$

  N4. $x + Sy = S(x + y)$

  N5. $x \cdot 0 = 0$

  N6. $x \cdot Sy = (x \cdot y) + x$

  N7. $\neg(x < 0)$

  N8. $x < Sy \leftrightarrow x < y \vee x = y$

  N9. $x < y \vee x = y \vee y < x$

6.1.2 A Brief History of Automated Proof

The automation of mathematical reasoning coincides with the emergence of the field of Artificial Intelligence (AI), whose early pioneers embarked on a program to (mechanically) simulate human problem solving.[7] By 1960, theorem provers for the full first-order predicate calculus had been implemented by Paul Gilmore [Gil60] and by Hao Wang [Wan60b, Wan60a] in the United States, and by Dag Prawitz [PPV60] in Sweden. Although this mechanization constituted an important proof of concept, the practical utility of the theorem provers was limited, due to the combinatorial explosion of the search space encountered in proofs of anything other than relatively simple theorems.

Following Shankar's exposition [Sha94], it is useful to distinguish three approaches in the subsequent development of automatic theorem proving and proof checking: resolution theorem provers, nonresolution theorem provers, and proof checkers. This discussion focuses solely on developments in Europe and the United States. There is also significant work in automated theorem proving in the region formerly known as the USSR and in the People's Republic of China. The Chinese have been particularly active in the area of decision procedures for geometrical applications [BB89, p. 27].

The first efficient mechanization of proof grew out of work done by Alan Robinson in the early 1960s and published in 1965 [Rob65]. Robinson combined procedures independently suggested by Davis and Putnam and by Prawitz to automate a significantly more efficient proof procedure for the first-order predicate calculus known as resolution. The key notion from Prawitz was *unification*, an algorithm that gives the unique, most general substitution that creates a complementary pair of literals P and $\neg P$. Resolution is a complete refutation procedure for first order logic (see Section 6.1.3.1.3). After its introduction in the mid 1960s, resolution was a focal point for activity in automated theorem proving, yielding numerous extensions and optimizations. By 1978,

---

[7] MacKenzie [Mac95] and Bläsius and Bürckert [BB89] provide interesting histories of post-Euclidean developments in automated reasoning.

Loveland's textbook on automated theorem proving documented some 25 variants of resolution [Mac95, p. 14]. Despite this considerable activity and a steady increase in computing power, the early resolution theorem provers suffered from the same limitation that had plagued the previous generation of mechanical proof procedures: the combinatorial explosion of the proof search space.

The 1970s also witnessed the emergence of logic programming, originally attributed to Kowalski and Colmerauer [Kow88]. Colmerauer and his colleagues implemented a specialized resolution theorem prover called Prolog (abbreviating the French "Programmation en Logique") that implemented Kowalski's procedural interpretation of Horn clause logic[8]. The result was a theorem proving system that could be used as a programming language [SS86].

Despite a decline in the 1970s due largely to disappointing performance, research in resolution theorem proving continued. Although unification remained the crucial algorithm, resolution provers added sophisticated heuristics, data structures, and optimizations to manage combinatorial explosion. The result has been increasingly efficient, powerful systems. In the 1970s and early 1980s, research in resolution theorem proving existed primarily at Argonne National Laboratory, where Robinson had originally been introduced to automatic theorem proving. Argonne's Aura (Automated Reasoning Assistant) and, more recently, Otter [WOLB92] systems have successfully proven not only known theorems, but also open conjectures in several fields of mathematics. In addition to Otter, current state-of-the-art resolution provers include SETHEO [LSB92] and PTTP [Sti86]. Paulson characterizes Otter, SETHEO, and PTTP as "automatic theorem proving at its highest point of refinement" [Pau97] and notes their extremely high inference rates, their efficient use of storage, and their ability to prove many of the toughest benchmark problems.

Resolution methods yield proofs that are not readily understood by humans. This perceived weakness, as well as the difficulty of combining resolution with nonlogical inference techniques such as induction, led researchers to pursue other approaches, including various levels of human interaction and a renewed interest in heuristics. In the 1970s, Woody Bledsoe and his colleagues at the University of Texas began work in "nonresolution theorem-proving," pursuing proof procedures that yielded more natural and powerful proofs for mathematical theorems, as well as heuristics (like those of the early AI pioneers) that produced human-like proofs. Although Bledsoe initially developed an automated prover for set theory that combined both resolution and heuristics, he later replaced resolution with a more "natural" procedure, augmented with a "limit heuristic" for calculus proofs, algebraic simplification, and linear inequality routines. The resulting prover successfully proved theorems in elementary calculus that had stymied existing resolution-style provers [Mac95].

Robert Boyer and J Strother Moore have collaborated on several influential theorem provers that use heuristics to develop proofs by induction and rewriting. The Nqthm series of provers [BM79,BM88], and its successor ACL2 [KM94,KM96], are highly automated, but require user guidance to accomplish difficult proofs. In the hands of skilled practitioners, the Boyer-Moore prover has been used to prove program and hardware correctness [BHMY89, Hun87], as well as mathematical theorems, including the automated proof of Gödel's incompleteness theorem undertaken by Shankar for his doctoral dissertation [Sha94].

---

[8] A clause is Horn if it has at most one positive literal, for example, $\neg P(x) \vee \neg Q(x) \vee R(x)$.

Other productive approaches to automatic theorem proving have included conditional rewriting as found in the Rewrite Rule Laboratory (RRL) system [KZ89] and *matings* as used by Andrews and his colleagues to develop a theorem prover for higher-order logic [AMCP84].

The distinction between theorem provers and proof checkers is tenuous, typically reflecting the intended use of the system or the degree of automation relative to other systems, rather than hard and fast differences.[9] Nevertheless, certain systems are more consistently identified as proof checkers. Automath, developed by de Bruijn and his colleagues at the Technische Hogeschool in Eindhoven, The Netherlands, was one of the earliest and most influential proof checkers, originating ideas subsequently used by several modern languages and inference systems [Sha94, p. 19]. Automath provided a grammar whose rules encoded mathematics in a way that allowed mechanized checks of correctness for Automath statements, as illustrated in [vBJ79].

The LCF (Logic for Computable Functions) system is another influential proof checker. In LCF, axioms are primitive theorems, inference rules are functions from theorems to theorems, and typechecking guarantees that theorems are constructed only by axioms and rules [Pau91, p. 11]. There are higher-order functions known as tactics and control structures known as tacticals (see Section 6.1.3.3), yielding a programmable system in which the user determines the desired level of automation. LCF has been used to verify program properties [GMW79] and to check the correctness of a unification algorithm [Pau84]. Several well-known systems have evolved from LCF, including HOL, Nuprl, and Isabelle. HOL (Higher-Order Logic) is a widely used system with extensive libraries that is employed primarily for verification of hardware and real-time systems. Nuprl is based on constructive type theory and was developed at Cornell University by Joseph Bates and Robert Constable as a mechanization of Bishop's program of constructively reconstructing mathematics [Sha94, p.19]. The Nuprl system has been used primarily as a research and teaching tool in the areas of constructive mathematics, hardware verification, software engineering, and computer algebra. Isabelle is a generic, interactive theorem prover based on the typed lambda calculus, whose primary inference rule is a generalization of Horn-clause resolution. Isabelle supports proof in any logic whose inference rules can be expressed as Horn clauses [Pau97]. Isabelle represents a synthesis between two largely distinct traditions in automated reasoning: resolution theorem proving and interactive theorem proving.

---

[9] For example, Shankar variously identifies both Nqthm [Sha94] and PVS [SOR93] as theorem provers and proof checkers.

6.1.3 Techniques Underlying Automated Reasoning

The preceding discussion identified major proving traditions including resolution, equational or rewrite systems, constructive type theory methods, Boyer-Moore-style systems, and a variety of other methods loosely characterizable as interactive. The resulting systems can be classified in various ways, including the interelated dimensions suggested by Gordon [Gor]: type of logic supported, extensibility, degree of automation, and closeness to underlying logic. Generic theorem provers can be configured for a variety of logics while specialized theorem provers exploit a particular application-oriented logic (for example, temporal logic model checkers) or contain features optimized for selected applications. There are several variations on extensibility; a theorem prover may not be extensible, or it may offer a metalogic (allowing the user to program the underlying logic), an extendable infrastructure (allowing the user to program sequences of proof steps), a reflective capability (allowing the prover to reason about its own soundness and thereby the soundness of proposed extensions), or a customizable syntax (ranging from alternative notations to parser support). In general, specialized systems such as model checkers are more automatic than general-purpose provers, all of which use some degree of automation. Degree of automation is also influenced by the closeness of proof to the underlying logic. Systems in which theorem proving differs little from the process of formal proof in the underlying logic tend to be more automated than those in which the difference is greater.

6.1.3.1 Calculi for First-Order Predicate Logic

In principle, inference rules may be used in one of two ways [BB89]. Starting from the logical axioms, inference rules may be applied until the formula to be proven (valid or unsatisfiable, depending on whether the calculus is positive or negative, respectively) is derived. This approach is called a *deductive calculus.* Alternatively, starting from the formula whose validity or unsatisfiability is to be shown, inference rules may be applied until logical axioms are derived. This second approach is termed a *test calculus.* The relationship between deductive and test calculi is analogous to that between forward and backward chaining state transition systems. As these remarks suggest, there is a variety of different calculi for first-order predicate logic, each offering a different perspective on the nature of validity [BE93]. The Gentzen calculus, including the variant known as the sequent calculus, is a positive deductive calculus, whereas Robinson's resolution calculus is a negative test calculus. These two calculi are introduced following a brief discussion of normal forms for predicate logic formulas. A survey of logical calculi may be found in [BE93].

6.1.3.1.1 Normal Forms

Normal forms are standardized formats intended to make predicate logic formulas easier to understand and manipulate. This section considers two such forms: *prenex normal form* and *skolem normal form.* Valid formulas of the form $A \Leftrightarrow B$, including important (tautological) equivalences such as the laws of quantifier distribution and the laws of quantifier movement, may be used (in conjuction with a variable renaming rule to avoid unintentional variable binding) as value-preserving transformations. These transformation rules yield a logically equivalent prenex form in which all quantifiers occur on the left, in front of the quantifier-free matrix [BB89]. *Skolemization*, named after the Norwegian mathematician Thoralf Skolem, yields a normal form that is particularly useful because it explicitly represents quantificational dependencies of assignments to variables. Following an explanation in [BB89], a formula $\forall x_1, \ldots, x_n \exists y \mathcal{F}$ in prenex form may be transformed into $\forall x_1, \ldots, x_n \mathcal{F}^*$, where $\mathcal{F}^*$ is obtained from $\mathcal{F}$ by replacing each free occurrence of y with a Skolem function, f_y, of the form $f_y(x_1, \ldots, x_n)$. The process of skolemization is not model-preserving, that is, a formula and its skolemized form are not equivalent. However, a formula is satisfiable (unsatisfiable) just in case its skolemized form is. Since only universal quantifiers remain after skolemization, the quantifiers are often implicitly assumed, yielding formulas of the form $\mathcal{F}^*$.

There are various skolemization strategies. The method described here begins with a formula in prenex form, but it is also possible to skolemize a formula that is not in prenex form by keeping track of the essential "parity" of the quantifier. Parity refers to the number of negations in whose scope the quantifier occurs. Almost all mechanical theorem provers use some form of skolemization.

6.1.3.1.2 The Sequent Calculus

The sequent calculus is a variant of the deductive calculus developed for his dissertation by the German logician Gerhard Gentzen [Gen70]. Gentzen was interested in using syntactic inference rules to model mathematical reasoning, and he defined the sequent calculus to make the assumptions on which a formula depended more transparent. This tranparency yields a calculus that is particularly suited to computer-assisted proof because the information relevant to a given part of the proof is localized. Two additional advantages attributed to the sequent calculus include the intuitively plausible nature of its inference rules and their symmetric construction, yielding relatively systematic and natural proof construction.

A sequent is written $\Gamma \vdash \Delta$, meaning $\wedge\Gamma \supset \vee\Delta$, where Γ is a (possibly empty) list of formulas $\{A_1, \ldots, A_m\}$ and Δ is a (possibly empty) list of formulas $\{B_1, \ldots, B_n\}$. In a sequent $\Gamma \vdash \Delta$, the formulas in Γ are called the *antecedents* and the formulas in Δ are called the *succedents* or *consequents*. Intuitively, the conjunction of the antecedents should imply the disjunction of the succedents, that is, $A_1 \wedge \ldots \wedge A_m \supset B_1 \vee \ldots \vee B_n$. A sequent calculus proof can be viewed as a tree of sequents whose root is a sequent of the form $\vdash A$, where A is the formula to be proved and the antecedent of the sequent is empty. The proof tree is generated by applying inference rules of the form

$$\frac{\Gamma_1 \vdash \Delta_1 \quad \cdots \quad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}\,\mathbf{N}$$

Intuitively, the rule named **N** takes a leaf node of a proof tree of the form $\Gamma \vdash \Delta$ and adds the n new leaves specified in the rule. If n is zero, that branch of the proof tree terminates.

The propositional inference rules consist of the *Propositional Axiom* and rules for conjunction ($\wedge$), disjunction ($\vee$), implication ($\supset$), and negation ($\neg$). The Propositional Axiom rule applies when the same formula appears in both the antecedent and succedent, corresponding to the tautology $(\Gamma \wedge A) \supset (A \vee \Delta)$, where Γ and Δ consist of the conjunction and disjunction, respectively, of their elements.

$$\frac{}{\Gamma,\ A \vdash A,\ \Delta}\ Ax$$

There are two rules for each of the propositional connectives and for negation, corresponding to the antecedent and consequent occurrences of these connectives. The negation rules simply state that the appearance of a formula in the antecedent is equivalent to the appearance of its negated form in the succedent, and vice versa.

$$\frac{\Gamma \vdash A, \Delta}{\Gamma,\ \neg A \vdash \Delta}\ \neg\vdash \qquad\qquad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A,\ \Delta}\ \vdash\neg$$

The inference rules $\neg \vdash$ and $\vdash \neg$ are often referred to as the rules for "negation on the left" and "negation on the right," respectively. Negation on the left rule can be derived as follows. Using the identity $(X \supset Y) \equiv (\neg X \vee Y)$, the antecedent can be written $\neg\Gamma \vee (A \vee \Delta)$, which is equivalent to $(\neg\Gamma \vee A) \vee \Delta$, and to $\neg(\neg\Gamma \vee A) \supset \Delta$. Invoking one of De Morgan's Laws $(\neg(X \vee Y) \equiv (\neg X \wedge \neg Y))$, $\neg(\neg\Gamma \vee A) \supset \Delta$ is equivalent to $(\Gamma \wedge \neg A) \supset \Delta$, which is an interpretation of the succedent.

The same symmetric formulation and naming conventions are used for the other rules, including those for the binary connectives. The rule for conjunction on the left is a consequence of the fact that the antecedents of a sequent are implicitly conjoined; the rule for conjunction on the right causes the proof tree to divide into two branches, requiring a separate case for each of the two conjuncts.

$$\frac{A,\ B,\ \Gamma \vdash \Delta}{A \wedge B,\ \Gamma \vdash \Delta}\ \wedge\vdash \qquad\qquad \frac{\Gamma \vdash A,\ \Delta \qquad \Gamma \vdash B,\ \Delta}{\Gamma \vdash A \wedge B,\ \Delta}\ \vdash\wedge$$

The rules for disjunction are duals of those for conjunction.

$$\frac{A,\ \Gamma \vdash \Delta \qquad B,\ \Gamma \vdash \Delta}{A \vee B,\ \Gamma \vdash \Delta}\ \vee\vdash \qquad\qquad \frac{\Gamma \vdash A,\ B,\ \Delta}{\Gamma \vdash A \vee B,\ \Delta}\ \vdash\vee$$

The rule for implication on the right is a consequence of the implication "built in" to the interpretation of a sequent. The rule for implication on the left splits the proof into two branches analogous to the two cases encountered with the rules for conjunction on the right and disjunction on the left. Note that one case of the implication on the left rule requires the antecedent to the implication be proved and the other case allows the consequent of the implication to be assumed.

$$\frac{\Gamma \vdash A,\, \Delta \qquad B,\, \Gamma \vdash \Delta}{A \supset B,\, \Gamma \vdash \Delta} \supset\vdash \qquad\qquad \frac{\Gamma,\, A \vdash B,\, \Delta}{\Gamma \vdash A \supset B,\, \Delta} \vdash\supset$$

To illustrate propositional reasoning in the sequent calculus, consider the following proof of

$$(P \supset Q \supset R) \supset (P \wedge Q \supset R).$$

reproduced from [Rus93b, pp. 231-233]. The implies symbol $\supset$ associates to the right and binds less tightly than $\wedge$. This formula is actually an instance of the law of exportation.

The first step is to contruct the goal sequent

$$\vdash (P \supset Q \supset R) \supset (P \wedge Q \supset R).$$

and then seek an applicable inference rule. There is only one choice in this case: the rule for implication on the right (with $[A \leftarrow (P \supset Q \supset R), B \leftarrow (P \wedge Q \supset R)]$ and Γ and Δ empty).

$$\frac{(P \supset Q \supset R) \vdash (P \wedge Q \supset R)}{\vdash (P \supset Q \supset R) \supset (P \wedge Q \supset R)} \vdash\supset$$

Considering the sequent above the line

$$(P \supset Q \supset R) \vdash (P \wedge Q \supset R)$$

there are two choices for the next step: implication on the left or implication on the right. Implication on the left will cause the proof tree to branch. Since it is usually best to delay branching as long as possible, implication on the right is the best option (this time with $[\Gamma \leftarrow (P \supset Q \supset R), A \leftarrow P \wedge Q, B \leftarrow R]$ and Δ empty)

$$\frac{(P \supset Q \supset R), (P \wedge Q) \vdash R}{(P \supset Q \supset R) \vdash (P \wedge Q \supset R)} \vdash\supset$$

Focusing once again on the sequent above the line

$$(P \supset Q \supset R), (P \wedge Q) \vdash R$$

there are two options: implication on the left or conjunction on the left. As in the last step, the strategy of delaying branching as long as possible narrows the choice. Applying conjunction on the left yields

$$\frac{(P \supset Q \supset R), P, Q \vdash R}{(P \supset Q \supset R), (P \wedge Q) \vdash R} \wedge\vdash$$

Now the sequent above the line is

$$(P \supset Q \supset R), P, Q \vdash R$$

and the only choice is to use the rule for implication on the left

$$\frac{(Q \supset R), P, Q \vdash R \qquad P, Q \vdash P, R}{(P \supset Q \supset R), P, Q \vdash R} \supset\vdash$$

The right branch can be closed immediately[10]

$$\frac{}{P, Q \vdash P, R} Ax$$

The left branch requires another application of the rule for implication on the left:

$$\frac{R, P, Q \vdash R \qquad P, Q \vdash Q, R}{(Q \supset R), P, Q \vdash R} \supset\vdash$$

The left and right branches can then be closed:

$$\frac{}{R, P, Q \vdash R} Ax$$

$$\frac{}{P, Q \vdash Q, R} Ax$$

Since all the branches are closed, the theorem is proved.

The preceding steps can be collected into the following "proof tree" representation:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{R, P, Q \vdash R} Ax \qquad \cfrac{}{P, Q \vdash Q, R} Ax}{(Q \supset R), P, Q \vdash R} \supset\vdash \qquad \cfrac{}{P, Q \vdash P, R} Ax}{(P \supset Q \supset R), P, Q \vdash R} \supset\vdash}{(P \supset Q \supset R), (P \wedge Q) \vdash R} \wedge\vdash}{(P \supset Q \supset R) \vdash (P \wedge Q \supset R)} \vdash\supset}{\vdash (P \supset Q \supset R) \supset (P \wedge Q \supset R)} \vdash\supset$$

First-order sequent calculus extends the propositional sequent calculus presented above with inference rules for universal and existential quantification and with an inference rule for nonlogical axioms.[11] In the statement of the quantifier rules, a is a new constant (that is, a constant that does not occur in the consequent of the sequent) and t is a term.

---

[10] Strictly speaking, it is first necessary to use an Exchange rule to reorder the formulas in the antecedent, and similarly for closure of the left branch, below. The Exchange rules are introduced at the end of this section.

[11] Technically, it is also convenient to modify the propositional axiom to allow not only for the case where the formula in the antecedent is the same as that in the consequent, but also for the case of two syntactically equivalent formulas, that is, formulas that are the same modulo the renaming of bound variables.

$$\frac{\Gamma,\ A[x \leftarrow t] \vdash \Delta}{\Gamma,\ (\forall x : A) \vdash \Delta}\ \forall\vdash \qquad \frac{\Gamma \vdash A[x \leftarrow a],\ \Delta}{\Gamma \vdash (\forall x : A),\ \Delta}\ \vdash\forall$$

$$\frac{\Gamma,\ A[x \leftarrow a] \vdash \Delta}{\Gamma,\ (\exists x : A) \vdash \Delta}\ \exists\vdash \qquad \frac{\Gamma \vdash A[x \leftarrow t],\ \Delta}{\Gamma \vdash (\exists x : A),\ \Delta}\ \vdash\exists$$

The quantifier rules are the sequent calculus analog of skolemization (cf. Section 6.1.3.1.1). The basic idea is that to prove a universally quantified formula, it is sufficient to show that the formula holds for an arbitrary constant (a), and to prove an existentially quantified formula, it is only necessary to show that the formula holds for a given term (t). The four quantifier rules reflect the underlying duality between universal and existential quantification.

The rule for nonlogical axioms is used to terminate a branch of the proof tree when a nonlogical axiom or previously proved lemma appears in the consequent of a sequent.

$$\frac{}{\Gamma \vdash A,\ \Delta}\ \mathit{Nonlog\text{-}ax}$$

where A is a nonlogical axiom or previously proved lemma.

For convenience in developing proofs, it is useful to provide an additional rule called "cut" as a mechanism for introducing a case-split or lemma into the proof of a sequent $\Gamma \vdash \Delta$ to yield the subgoals $\Gamma, A \vdash \Delta$ and $\Gamma \vdash A, \Delta$. The subgoals are equivalent to assuming A along one branch and having to prove it on the other. Alternatively, applying the rule for negation on the right, the subgoals are equivalent to assuming A along one branch and $\neg A$ along the other.

$$\frac{A,\ \Gamma \vdash \Delta \qquad \Gamma \vdash A,\ \Delta}{\Gamma \vdash \Delta}\ \mathit{Cut}$$

The Cut rule can be omitted; a well-known result in proof theory, the Cut Elimination Theorem (also known as Gentzen's Hauptsatz), establishes that any derivation involving the cut rule can be converted to another (possibly much longer proof) that does not use cut. Since cut is the only rule in which a formula (A) appears above the line that does not also appear below it, it is the only rule whose use requires "invention" or "insight"; thus, the cut elimination theorem provides the foundation for another demonstration of the semi-decidability of the predicate calculus [Rus93b, p. 244].

Finally, there are four structural rules that simply allow the sequent to be rearranged or weakened. These rules have the same status as the Cut rule; they can also be omitted. The Exchange rules allow formulas in the antecedent and consequent to be reordered.

$$\frac{\Gamma_1,\ B,\ A,\ \Gamma_2 \vdash \Delta}{\Gamma_1,\ A,\ B,\ \Gamma_2 \vdash \Delta}\ X\vdash \qquad \frac{\Gamma \vdash \Delta_1,\ B,\ A,\ \Delta_2}{\Gamma \vdash \Delta_1,\ A,\ B,\ \Delta_2}\ \vdash X$$

The Contraction rules allow multiple occurrences of the same sequent formula to be replaced by a single occurrence.[12]

$$\frac{A,\ A,\ \Gamma \vdash \Delta}{A,\ \Gamma \vdash \Delta}\ C\vdash \qquad\qquad \frac{\Gamma \vdash A,\ A,\ \Delta}{\Gamma \vdash A,\ \Delta}\ \vdash C$$

6.1.3.1.3 The Resolution Calculus

The resolution calculus is a negative test calculus for formulas in clausal form[13]; it contains a single logical axiom and uses only one rule of inference, the *resolution rule*. The single axiom is an elementary contradiction denoted by the empty clause (□). In its simplest form, the resolution rule may be viewed as a special instance of the cut rule (of the sequent calculus) in which all single formulas are literals [BB89, p. 56]. Using notation from [BB89], the resolution rule is defined as follows.

$$\begin{array}{lll} \mathit{clause}1: & L, & K_1, \ldots, K_n \\ \mathit{clause}2: & \neg L, & M_1, \ldots, M_m \\ \hline \mathit{resolvent}: & K_1, \ldots, K_n & M_1, \ldots, M_n \end{array}$$

where clauses 1 and 2 are referred to as the *parent clauses* of the resolvent and L and $\neg L$ are the *resolution literals.*

A generalization of this rule allows an instantiation of the formulas in terms of a *substitution* that maps variables to terms uniformly across both resolution literals. Using the same notation, the general resolution rule is expressed as shown below, where σ represents a substitution that makes the atoms L and L' equal, that is, $\sigma L = \sigma L'$.

$$\begin{array}{lll} \mathit{clause}1: & L, & K_1, \ldots, K_n \\ \mathit{clause}2: & \neg L', & M_1, \ldots, M_m \\ \hline \mathit{resolvent}: & \sigma K_1, \ldots, \sigma K_n & \sigma M_1, \ldots, \sigma M_n \end{array}$$

If a substitution, σ exists for two expressions, the expressions are said to be *unifiable* and the substitution is called a *unifier* for the two expressions. Given a pair of expressions, there are distinguished unifiers, known as the *most general unifiers* from which all other unifiers may be derived by instantiation.

The following examples are taken from [BB89]. The terms x and $f(y)$ are unifiable. The substitution $\{x \leftarrow f(y), y \leftarrow a\}$ is a unifier for x and $f(y)$, although it is not a most general one, since it can be obtained from the most general unifier $\{x \leftarrow f(y)\}$ by further instantiating y with a. There are two equivalent most general unifiers for the pair of terms $f(x, g(x))$ and $f(y, g(y))$: $\{x \leftarrow y\}$ and $\{y \leftarrow x\}$, which differ only with respect to variable names. The terms x and $f(x)$ are not unifiable. Neither are the terms $g(x)$ and $f(x)$.

---

[12] The structural rules (Contraction and Exchange) are sometimes formulated in terms of a single weakening rule.

[13] That is, a disjunction of literals, where a literal is a proposition or a negated proposition. Quantifiers are not permitted. Universal quantifiers are implicit, and existential quantifers are replaced by Skolem functions as described in Section 6.1.3.1.1. For example, in clausal form, $\forall x \exists y R(x, y)$ becomes $R(x, f(x))$.

Resolution is a complete refutation procedure for first-order logic. If a sentence is false under all interpretations over all domains—that is, *unsatisfiable*, then resolution will terminate with the empty clause indicating a contradiction has been derived. If the negation is unsatisfiable, the original theorem is true. If the original theorem is not true, resolution may not terminate. The proof of the completeness of resolution is based on a result from Jacques Herbrand's 1930 dissertation. Roughly, Herbrand's theorem states that in proving a set of clauses, S, unsatisfiable, the only substitutions that need to be tried are those drawn from the set, H, of all variable-free terms formed from the functions (including constant functions) occurring in S. The set, H, is known as the *Herbrand Universe* of S. Since H is always either finite or countably infinite, a contradiction, if one exists, will always be found [CL73].

Resolution theorem provers can be highly effective in some domains. In general, they have not been used in formal methods because it has been difficult to combine resolution with induction and with the additional first-order theories necessary for formal methods applications. Furthermore, resolution methods do not readily support proof exploration and typically yield proofs that are not easily understood by humans. Nevertheless, fundamental techniques from resolution-based provers, such as highly efficient unification algorithms, have been incorporated in most modern theorem provers.

6.1.3.2 Extending the Predicate Calculus

The predicate calculus is not sufficient for most applications of formal methods, which typically require the addition of first-order theories such as equality, arithmetic, simple computer science datatypes (lists, trees, arrays, and so forth), and set theory. These four theories are basic to most applications. Particular applications may benefit (significantly) from the inclusion of additional first-order theories. Formal methods also require support for induction. In general, methods for automating inductive proofs are less well-developed, and user guidance is typically required for such proofs. A discussion of the current status of automated induction and the role of induction in formal methods appears in [Rus96]. The discussion includes an interesting fragment drawn from a specification of Byzantine fault-tolerant clock synchronization.

The next three sections summarize some of the issues involved in developing first-order theories for equality and arithmetic, and introduce the topic of combinations of theories.

6.1.3.2.1 Reasoning about Equality

The fundamental notion of equality is that if x and y are equal, then x and y have all properties in common, that is, $x = y$ if and only if, x has every property that y has and, conversely, y has every property that x has. This idea was first formulated by Leibniz and is also referred to as "Leibniz's Law" [Tar76, p. 55]. Equality is reflexive, symmetric, and transitive and is therefore an equivalence relation. However, equality also satisfies the property of substitutivity (that is, equals may always be substituted for equals) and is thereby distinguished from mere equivalence relations.

A model for a first-order theory with equality that interprets "=" as the identity relation on the domain of interpretation is referred to as a *normal* model. Since it is possible to show that a first-order system with equality has a model if and only if it has a normal model, nothing is lost by restricting the focus to a normal model. An *initial* model is one without "confusion" or "junk," where confusion and junk may be informally defined as the ability to assign elements to terms in a way that simultaneously preserves as distinct those terms not required to be equal by the axioms (no confusion) and leaves no element unassigned (no junk).

The sequent calculus rules for equality directly encode the axiom for reflexivity (that states that everything equals itself) and Leibniz's rule. The rules of transitivity and symmetry for equality can be derived from these rules. The notation $A[e]$ denotes occurrences of e in A in which no free occurrences of variables of e appear bound in $A[e]$ and, similarly, for $\Delta[e]$. $mathbfRefl$ additionally requires that a and b be syntactically equivalent, that is, $a \equiv b$,

$$\frac{}{\Gamma \vdash a = b, \Delta}\ \textit{Refl} \quad \text{if } a \equiv b \qquad\qquad \frac{a = b, \Gamma[b] \vdash \Delta[b]}{a = b, \Gamma[a] \vdash \Delta[a]}\ \textit{Repl}$$

Reasoning about equality is so basic that most theorem provers use very efficient methods to handle the chains of equality reasoning that invariably arise in automated theorem proving. Examples include efficient algorithms for computing the *congruence closure* relation on the graph representing the terms in a ground formula[14] [Sho78b, DST80, NO79].

Equations also commonly arise in the form of definitions, such as that for the absolute value function[15]:

$$|x| = \textbf{if } x < 0 \textbf{ then } -x \textbf{ else } x.$$

One way to prove a theorem such as $|a + b| \leq |a| + |b|$ is to expand the definitions and then perform propositional and arithmetic reasoning. "Expanding a definition" involves finding a substitution for the left side of the definition that causes it to match a term in the formula (for example, $[x \leftarrow a + b]$ will match $|x|$ with $|a + b|$), and then replacing that term by the corresponding substitution instance of the right side of the given definition—for example,

$$|a + b| = \textbf{if } a + b < 0 \textbf{ then } -(a + b) \textbf{ else } a + b.$$

Expanding definitions is a special case of the more general technique of *rewriting*. "The basic idea is to impose directionality on the use of equations in proofs; ... directed equations are used to compute by repeatedly replacing subterms of a given formula with equal terms until the simplest form possible is obtained." [DJ90] The notion of directed equation refers to the fact that although equations are symmetric—$a = b$ means the same thing as $b = a$—rewriting imposes a left-to-right orientation, hence equations viewed with this orientation are generally called *rewrite rules*. Rewriting may be used with arbitrary equations provided the free variables appearing on the right side of each equation are a subset of those appearing on its left. The process of identifying substitutions for the free variables in the left side of the formula of interest as a prerequisite to rewriting is called *matching*. Matching is sometimes referred to as "one way" unification; although the process is essentially the same, only substitutions for the variables in the equation being matched are of interest.

---

[14] A ground formula is one in which there are no occurrences of free variables.

[15] This and the following example are reproduced from [Rus93b].

Rewriting may be automated or performed by the user. Two desirable properties of rewrite rules are *finite termination* and *unique termination*, also known as Church-Rosser. A set of rewrite rules has the finite termination property if rewriting always terminates. A set of rewrite rules is Church-Rosser if the final result after rewriting to termination is independent of the order in which the rules are applied. There are effective heuristic procedures for testing for the finite and unique termination properties, including Knuth-Bendix completion [KB70, DJ90], which can often be used to extend a set of rewrite rules that is not Church-Rosser into one that is. A theory that can be specified by a set of rewrite rules with both the finite and unique termination properties may be used as a decision procedure for that theory. Moreover, any such decision procedure is sound and, for ground formulas, complete. However, deducing disequalities is sound and complete only for the initial model. Therefore, systems that use rewriting to normal form as their primary or only means of deduction typically use an initial model semantics, whereas systems that use rewriting in conjuction with other methods typically use a classical semantics and (only) infer disequalities axiomatically.

There are several variations on rewriting, including order-sorted rewriting, conditional rewriting, priority rewriting, and graph rewriting. A description of these variants appears in the comprehensive survey of rewrite systems provided in [DJ90].

Term rewriting is highly effective, and essential to the productive use of theorem proving for formal methods applications. It serves as the primary means of deduction in Affirm [GMT$^+$80, Mus80], Larch [wSJGJMW93], and RRL [KZ89] and is one of the most important techniques in the Boyer-Moore provers [BM88, KM96]. Rewriting may also be used for computation [GKK$^+$88]. The paramodulation techniques [RW69] used in resolution are similar to rewriting.

6.1.3.2.2 Reasoning about Arithmetic

Formal methods applications typically involve arithmetic expressions and relations over both real and natural numbers, and both interpreted and uninterpreted function symbols. The ubiquity and often the complexity of this arithmetic make efficient deductive support for arithmetic essential to the productive use of formal methods [Rus96]. Integer arithmetic is sufficiently important that some formal methods systems include decision procedures for the quantified theory of integer arithmetic, often referred to as Presburger arithmetic after the Polish mathematician who first studied these arithmetics in the late 1920s. The decidable fragment essentially includes linear arithmetics with addition, subtraction, multiplication, equality, the "less than" predicate ($<$), and, by simple constructions, the predicates $>$, $\leq$, $\geq$. Classic Presburger arithmetic, which contains neither function symbols nor anything other than simple constants, is decidable. However, given the importance of function symbols and the fact that they may be introduced into formulas in which they do not originally appear through the process of skolemization, it is easy to appreciate the tension between efficiency (decidability) and expressiveness. Tools for formal methods often opt to restrict the arithmetic decision procedures to the ground (that is, propositional) case, where the combination of linear arithmetic with uninterpreted function symbols is decidable [CLS96].

6.1.3.2.3 Combining First-Order Theories

One of the challenges in designing a truly useful theorem prover or proof checker is combining decidable theories both with one another and with user interaction. There are algorithms dating back to the late 1970s for combining decision procedures, including Nelson-Oppen [NO79] and Shostak [Sho78b, CLS96]. The Nelson-Oppen algorithm combines decision procedures for two disjoint ground theories (for example, linear arithmetic and lists) by "introducing variables to name subterms and iteratively propagating any deduced equalities between variables from one theory to another" [CLS96]. Shostak combines theories that are canonizable (that is, can be converted to a canonical or normal form) and algebraically solvable using an optimized implementation of the congruence closure algorithm for ground equality over uninterpreted function symbols [CLS96]. Since Shostak's approach appears to be very efficient, but is restricted to algebraically solvable theories, there is some interest in combining it with Nelson-Oppen.

Most automated theorem provers and proof checkers minimally contain implementations of decision theories for propositional logic, equality, and linear arithmetic. Cyrluk et al. note that the method used to combine decision procedures is more critical to the overall efficiency of the system than the efficiency of any single decision procedure [CLS96]. Therefore, it is important that new decision procedures, such as those that arise in response to needs identified in formal methods applications, work effectively in combination with other theories.

6.1.3.3 Mechanization of Proof in the Sequent Calculus

This section illustrates how the proof of $(P \supset Q \supset R) \supset (P \wedge Q \supset R)$ from Section 6.1.3.1.2 might proceed with the help of an interactive theorem prover built on the sequent calculus style of reasoning. The presentation assumes that the formula has been introduced to the toolset and given the name "`theorem_1`." Invoking a proof attempt on this theorem places the user at the theorem prover's interactive interface. Intermediate sequents developed during the course of the proof are displayed, allowing the user to guide the proof at each step.

```
Beginning proof of "theorem_1":

    Antecedents:
          None
      =======>
        Consequents:
          Formula 1:  (P => (Q => R)) => ((P & Q) => R)
```

The imaginary prover used in this example displays sequents in the format shown above. Actual provers use similar formats, although they are usually less verbose.

The theorem prover's interaction style is based on the user's entry of a command to invoke a proof step, and the prover's display of the results of that command. Application of inference rules is the main type of command, with various supporting utility functions, such as proof status and proof display commands, provided as well.

```
    Step 1: apply-rule "implies-right"

    Applying rule "implies-right" to the current sequent yields:

        Antecedents:
          Formula 1:  P => (Q => R)
      =======>
        Consequents:
          Formula 1:  (P & Q) => R
```

The user would type the command after the "`Step 1:`" prompt shown above. In this case, the inference rule causes a new sequent to be generated, as shown.

```
    Step 2: apply-rule "implies-right"

    Applying rule "implies-right" to the current sequent yields:

        Antecedents:
          Formula 1:  P => (Q => R)
          Formula 2:  P & Q
      =======>
        Consequents:
          Formula 1:  R
```

A second application of the "`implies-right`" rule creates multiple antecedents. Depending on the formulas involved, this might mean future commands could apply to more than one formula. The prover would have to pick one by default or require the user to specify the formula.

```
    Step 3: apply-rule "and-left"
```

```
Applying rule "and-left" to the current sequent yields:

    Antecedents:
      Formula 1:  P => (Q => R)
      Formula 2:  P
      Formula 3:  Q
  =======>
    Consequents:
      Formula 1:  R
```

At this point, the proof has progressed without branching, but case splitting will now be required. The next rule application causes branching in the proof tree, for which the prover supplies suitable node numbers to keep track of current and future locations.

```
Step 4: apply-rule "implies-left"

Applying rule "implies-left" to the current sequent produces
two cases, the first of which, case 4.1, is as follows:

    Antecedents:
      Formula 1:  Q => R
      Formula 2:  P
      Formula 3:  Q
  =======>
    Consequents:
      Formula 1:  R
```

Only one branch can be pursued at a time. The prover will automatically return to the second branch after the current one is completed.

```
Case 4.1, step 1: apply-rule "implies-left"

Applying rule "implies-left" to the current sequent produces
two cases, the first of which, case 4.1.1, is as follows:

    Antecedents:
      Formula 1:  R
      Formula 2:  P
      Formula 3:  Q
  =======>
    Consequents:
      Formula 1:  R
```

A second application of the same rule causes further case splitting. Systematic navigation of the proof tree helps the user keep his or her bearings.

The current branch can be terminated by applying the Propositional Axiom that acknowledges when the sequent is a tautology. Normally, a user need not explicitly invoke this rule; most provers will recognize such opportunities and apply the rule automatically.

```
Case 4.1.1, step 1: apply-rule "prop-axiom"

Applying rule "prop-axiom" to the current sequent completes the
proof of case 4.1.1.

Resuming with case 4.1.2:

    Antecedents:
      Formula 1:  P
      Formula 2:  Q
  =======>
    Consequents:
      Formula 1:  Q
      Formula 2:  R
```

After dispensing with one branch of case 4.1, the prover presents the user with the other branch, and the “`prop-axiom`” rule applies again.

```
Case 4.1.2, step 1: apply-rule "prop-axiom"

Applying rule "prop-axiom" to the current sequent completes the
proof of case 4.1.2.  This also completes the proof of case 4.1.

Resuming with case 4.2:

    Antecedents:
      Formula 1:  P
      Formula 2:  Q
  =======>
    Consequents:
      Formula 1:  P
      Formula 2:  R
```

After finishing all of case 4.1, the prover pops back up to case 4.2 to finish off the only remaining branch of the proof.

```
Case 4.2, step 1: apply-rule "prop-axiom"

Applying rule "prop-axiom" to the current sequent completes the
proof of case 4.2.  This also completes the proof of "theorem_1".

Q.E.D.
```

Recognizing that all branches of the tree have resulted in valid proofs, the prover announces the successful completion of the overall proof.

Although this example has been cast in terms of a fictitious theorem prover, many actual provers follow a similar style of interaction. Level of automation, and therefore the nature of the interaction, varies considerably from one prover to another. For example, the level of automation may make it unnecessary to invoke rules at the level of detail presented here. Although the prover typically builds the full proof tree, the user generally sees only those portions of the tree requiring user guidance. Trivial cases are typically not displayed, although there may be a facility for revisiting both the implicit and explicit steps of a proof. On first attempt, most putative theorems attempted are incorrect, that is, they are not in fact theorems. Therefore it is at least as important that an automated deduction system facilitate the discovery of error, as that it should efficiently prove true theorems.

In addition to mechanizing routine manipulations, automated deduction systems should reduce the low-level interaction and repetitive tedium involved in large proofs. To this end, many interactive provers provide higher-order functions known as *tactics* and control structures known as *tacticals* that allow simple tactics to be combined into more complex ones. Paulson [Pau92, p. 456] notes that ideally tactics should capture the control structures typically used in describing proofs. He also remarks that, in practice, tactics often do not work at the level of proof description, but rather at a somewhat lower level. Nevertheless, tacticals potentially allow the user to perform hundreds of inferences with a single command. The concept and implementation of tactics and tacticals varies from prover to prover, but all share the idea that a theorem prover should be programmable. The challenge in using any automated reasoning system is to learn to use the automation effectively by exploiting the system's strengths and realizing its limitations.

Given the number and diversity of automated reasoning systems, the question invariably arises as to which system is most appropriate for a given application. Young [You95] suggests the use of benchmark problems to facilitate comparison of system performance within specific areas. Although standard benchmarks have yet to be identified, there are problems, such as the railroad gate controller [HL94] in the area of safety-critical systems, that have been attempted on a variety of systems.

6.1.4 Utility of Automated Deduction

Deductive techniques support more varied and more abstract models, more expressive specification, more varied properties, and more reusable verifications than finite state verification techniques. The essential utility of theorem proving or proof checking includes the following. With the exception of establishing the consistency of axioms, these benefits are self-explanatory and are listed with little additional comment.

- *Guarantee Type Correctness:* The type correctness of some specification languages is undecidable. In such cases, theorem proving or proof checking may be used to discharge obligations incurred during typechecking, thereby establishing that a specification is type-correct. As noted in Chapter 5, significant benefit accrues from typechecking alone.

- *Establish (Relative) Consistency of Axioms:* A specification can be shown to be consistent by demonstrating that its axioms have a model. In the context of mechanical verification, this is accomplished via theory intepretations that relate a source specification (the one to be shown consistent) to a target specification whose consistency has presumably already been established[16]. This is accomplished by defining a mapping from the types and constants of the source specification to those of the target specification and proving that the axioms of the source specification expressed in terms of that mapping are provable theorems of the target specification. The proof demonstrates *relative* consistency; that is, if the target specification is consistent, the source specification is consistent. [ORSvH95, p. 111] cites an example that underscores the need to validate axiomatic specifications by exhibiting an intended model.

- *Challenge Underlying Assumptions:* One of the benefits of formal specification is the explicit statement of underlying assumptions. Once formalized, these assumptions may be challenged by formulating and proving conjectures that exercise them. Design implications of new or modified assumptions may be similarly probed in this way.

- *Establish the Correctness of Hierarchical Layers:* Theory interpretations may also be used to demonstrate the correctness of hierarchical development, that is, to prove that a more concrete specification is a satisfactory implementation of a more abstract one, as illustrated in [Bev89, BDH94]. The approach is similar to that previously mentioned in the context of establishing the consistency of a set of axioms.

- *Confirm Key Properties and Invariants:* System properties and constraints may be precisely stated and deductively verified.

---

[16] For example, by specifying the target specification definitionally in a system that guarantees conservative extension.

- *Predict and Calculate System Behavior:* System behavior may be predicted or calculated by formulating and proving challenges or putative theorems that characterize the behavior or functionality of interest.

- *Facilitate Replication and Reuse:* Reusing and adapting extant proofs, as well as formulating new challenges, provides a systematic exploration of the implications of changes and extensions, as well as an effective vehicle for generalizing results for later reuse. Automation is the key to faithfully replicating or reusing a detailed deduction. The LaRC bit vectors library [BMS$^+$96] illustrates many of the issues involved in developing general and effectively reusable formal analyses.

The use of proof as a form of absolute guarantee is not included in this list for reasons outlined in Section 7.5, and succinctly captured in the following quote from [RvH93]: "A mechanical theorem prover does not act as an oracle that certifies bewildering arguments for inscrutable reasons, but as an implacable skeptic that insists on all assumptions being stated and all claims justified."

6.2 Finite-State Methods

The state space of a system can be loosely defined as the full range of values assumed by the state variables of the program or specification that describes it. The behaviors that the system can exhibit can then be enumerated in terms of this range of values. If the state space is finite and reasonably small, it is possible to systematically enumerate all possible behaviors of the system. However, few interesting systems have tractable state spaces. Furthermore, the state space of a formal specification can be infinite, for example, if it uses true mathematical integers as values for state variables. Nevertheless, there are various techniques for "downscaling" or reducing the state space of a system, while preserving its essential properties. Finite-state methods refer to techniques for the automatic verification of these finite-state systems or of infinite-state systems that can be similarly "reduced" by virtue of certain structural symmetries or uniformities. Given a formula specifying a desired system property, these methods determine its truth or falsity in a specific finite model (rather than proving its validity for all models). For linear-time and branching-time logics, the model checking problem is computationally tractable, whereas the validity problem is intractable.

6.2.1 Background

This section provides background information useful for an understanding of finite-state systems, including a brief introduction to temporal logics, fixed point characterizations, and the modal mu-calculus.

6.2.1.1 Temporal Logic

Temporal logic (also known as *tense* logic) [Pnu77,Bur84] augments the standard operators of propositional logic with *tense operators* that are used to formalize time-dependent conditions. The simplest temporal logic adds just two operators: the (weak) future operator, F, and the (weak) past operator, P. The formula Fq is true in the present if q is true at some time in the future and, similarly, the formula Pq is true in the present if q is true at some time in the past. These operators can be combined to assert quite complex statements about time-dependent phenomena. For example, $q \Rightarrow FPq$ can be interpreted as "if q holds in the present, than at some time in the future q will have held in the past." [McM93, p. 13] The duals of these operators, $\neg F \neg$, usually abbreviated G and $\neg P \neg$, usually abbreviated H, yield the corresponding strong future and past operators. $Gq \equiv \neg F \neg q$ means that q is true at every moment in the future, and $Hq \equiv \neg P \neg q$ means that q is true at every moment in the past.

A temporal logic system consists of a complete set of axioms and inference rules for proving all valid statements in the logic relative to a given model of time. Some of the more commonly used models include partially ordered time, linearly ordered time, discrete time, and branching (nondeterministic) time. Linear time corresponds to commonly held notions of time as a linearly ordered set measured with either the real or natural numbers. Discrete time refers to a model in which time is represented as a discrete sequence measured by the integers, as commonly found in engineering. Interval Temporal Logic [Mos85] is based on discrete time. Branching time is a model in which the temporal order $<$ defines a tree that branches toward the future; every instant has a unique past, but an indeterminate future [McM93, p. 15]. Temporal logic and the closely related *dynamic logic*[17] have been used to express program properties such as termination, correctness, safety, deadlock freedom, clean behavior, data integrity, accessibility, responsiveness, and fair scheduling [Bur84, p. 95]. Duration Calculus [CHR92], a notation used to specify and verify real-time systems, is an extension of interval temporal logic that uses a notion of *durations* of states within a time interval, but without explicit mention of absolute time. Temporal logics, and modal logics in general, are typically given a model theoretic semantics known as *possible worlds semantics*. A model in this semantics is usually referred to as a Kripke model, after Saul Kripke, one of the first mathematicians to give a model-theoretic interpretation of modal logic [Kri63a, Kri63b, Kri65]. The basic idea of Kripke semantics is to relativize the truth of a statement to temporal stages or states. Accordingly, a statement is not simply true, but true at a particular state. The states are temporally ordered, with the type of temporal order determined by the choice of axioms.

---

[17] The term "dynamic logic" refers generically to logical systems used to reason about computer programs. The basic premise is that certain classical logical systems that are inherently "static" can be extended quite naturally to reason about "dynamics." In addition to its application to computational systems, the study of dynamic logic and related topics has more general philosophical and mathematical implications as a natural extension of modal logic to general dynamic situations [Har84].

For example, the so-called *minimal tense logic*, K_t, is defined by van Benthem as follows [vB88, p. 7].

- Axioms:
    1. all propositional tautologies
    2. $G(\phi \rightarrow \psi) \rightarrow (G\phi \rightarrow G\psi)$
    3. $H(\phi \rightarrow \psi) \rightarrow (H\phi \rightarrow H\psi)$
    4. $\phi \rightarrow HF\phi$
    5. $\phi \rightarrow GP\phi$
- Definition:
    1. $F\phi \leftrightarrow \neg G \neg \phi$
    2. $P\phi \leftrightarrow \neg H \neg \phi$
- Rules of Inference:
    1. $\phi, \phi \rightarrow \psi / \psi$ (Modus Ponens)
    2. if ϕ is a theorem, then so are $G\phi$, $H\phi$ ((Temporal) Generalization)

Various axioms may be added to K_t to characterize further assumptions on the temporal order, such as transitivity and antisymmetry (which together yield a partial order), as well as density, linearity, and so forth. In the context of finite state methods, the notions of linear time and branching time are of particular interest.

The set of states in an interpretation represents not only past states, but all accessible (*possible*) future states. Furthermore, truth is persistent. Intuitively, this means that a sentence true at a given state will always be true at later states. The following definitions are due to Burgess [Bur84, pp. 93-4]. A *Kripke frame* is composed of a nonempty set S, equipped with a binary relation R. A *valuation* in a frame (S, R) is a function, V, that assigns to each variable, p_i, a subset of S, and each (syntactically well-formed) sentence a truth value. Intuitively, S represents the set of states and R represents the earlier-later relation. A formula, α is *valid* in a frame (S, R) if $V(\alpha) = X$ for every valuation V in (X, R). α is *satisfiable* in (X, R) if $V(\alpha) \neq \emptyset$[18] for some valuation V in (S, R), or, equivalently if $\neg\alpha$ is not valid in (S, R). In addition, α is *valid* over a class, $\mathcal{K}$, of frames if it is valid in every $(S, R) \in \mathcal{K}$, and is *satisfiable* over $\mathcal{K}$ if it is satisfiable in some $(S, R) \in \mathcal{K}$ or, equivalently, if $\neg\alpha$ is not valid over $\mathcal{K}$.

The interaction of universal and existential quantification with temporal operators is complex, introducing both philosophical and technical difficulties. Burgess [Bur84,

---

[18] $\emptyset$ denotes the empty set.

p. 131] notes that the philosophical issues include “identity through changes, continuity, motion and change, reference to what no longer exists or does not exist, essence and many, many more” and the technical issues include “undecidability, nonaxiomatizability, undefinability or multidimensional operators, and so forth.” Thoughtful discussion of these issues can be found in [Gar84] and [Coc84].

6.2.1.2 Linear Temporal Logic (LTL)

Linear time corresponds to the usual notion of time as a linearly ordered set, measured either with the real or the natural numbers. The temporal order relation $<$ is total, that is, antisymmetric, transitive, and comparable. Comparability means that for all states s_1 and s_2 in the same execution sequence, either $s_1 < s_2$ or $s_2 < s_1$ or $s_1 = s_2$). The extension of K_t obtained by adding the following two axioms (of right- and left-linearity, respectively) characterizes the linear temporal frames.

1. $(F\phi \wedge F\psi) \rightarrow F(\phi \wedge F\psi) \vee F(\phi \wedge \psi) \vee F(\psi \wedge F\phi)$
2. $(P\phi \wedge P\psi) \rightarrow P(\phi \wedge P\psi) \vee P(\phi \wedge \psi) \vee P(\psi \wedge P\phi)$

Alternatively, the following, somewhat more intuitive axioms can be used to characterize total orders [Bur84, p. 104].

1. $(FP\phi) \rightarrow (P\phi \vee \phi \vee F\phi)$
2. $(PF\phi) \rightarrow (P\phi \vee \phi \vee F\phi)$

Linear temporal logic is typically extended by two additional operators, the *until* operator and the *since* operator, abbreviated U and S, respectively.

The following definitions are based on a discussion in [McM93, p. 14] and assume that all subscripted states, $s_{..}$, are comparable. $\phi U \psi$ is true in state s_j if there is some state s_k such that $s_j < s_k$ and ψ is true in s_k, and for all s_i, such that $s_j < s_i < s_k$, ϕ is true in state s_i. Intuitively, ψ holds at some time in the future, until which time ϕ holds. Similarly, $\phi S \psi$ is true in state s_j just in case there is some state s_k such that $s_k < s_j$ and ψ is true in s_k, and for all s_i, such that $s_k < s_i < s_j$, ϕ is true in state s_i. Informally, ψ held at some time in the past, since which time ϕ has held.

6.2.1.3 Branching Time Temporal Logic

A treelike or branching frame is one in which the temporal order defines a tree that branches toward the future. Treelike frames represent ways in which things can evolve nondeterministically; every moment or state has a unique, linearly ordered past, but an indeterminate future. Following Thomason [Tho84, p. 142], a treelike frame for a tense logic consists of a pair $\langle T, < \rangle$, where T is a nonempty set and $<$ is a transitive ordering on T such that if $t_1 < t$ and $t_2 < t$, then either $t_1 = t_2$ or $t_1 < t_2$ or $t_2 < t_1$. The tree-ordered frames can be characterized by dropping the axiom

$$(PF\phi) \rightarrow (P\phi \vee \phi \vee F\phi)$$

from the axioms of linear time logic. A *branch* through $t \in T$ is a maximal linearly ordered subset of T containing t.

The semantics for branching time temporal logic are somewhat problematic. As Thomason [Tho84, p. 142] notes, interpreting future tense in these treelike structures can be perplexing. For example, take a simple structure with three moments, the root, t_0, and two branches labeled t_1 and t_2, respectively. Assume ϕ true at t_0 and t_1 and false at t_2. Is $F\phi$ true at t_0? It is hard to say. The answer involves technical issues that revolve around the reconciliation of tense with indeterminism. The logical argument for determinism claims that it is not possible to provide a correct definition of satisfaction for these structures, that is, to provide a definition that does not generate validities that are incompatible with the intended interpretation. Thomason [Tho84] presents an interesting discussion of strategies advanced by indeterminists to circumvent these claims.

The propositional branching time temporal logics that provide the foundation for one of the principal approaches to finite state verification of concurrent systems are called Computational Tree Logics. There are basically two variants: CTL and CTL*. The logic CTL* combines both branching-time and linear-time operators. A (computational) path quantifier, either A or E, denoting all or some paths, respectively, can prefix assertions composed of arbitrary combinations of the linear time operators G, F, U, and the "nexttime" operator, X (see below). There are two types of formulas in CTL*: *state formulas* that are true in a given state and *path formulas* that hold along a given path. The following definitions are taken from [CGK89, pp. 83-84]. Let AP be the set of atomic proposition names.
A state formula is either:

- A, if $A \in AP$.
- If f and g are state formulas, then $\neg f$ and $f \vee g$ are state formulas.
- If f is a path formula, then $\mathbf{E}(f)$ is a state formula.

A path formula is either:

- A state formula.

- If f and g are path formulas, then $\neg f$, $f \vee g$, $\mathbf{X}f$ and $f\mathbf{U}g$ are path formulas.

CTL* is the set of formulas generated by the above rules. CTL is a restricted subset of CTL* that permits only branching-time operators. CTL is obtained by limiting the syntax for path formulas to the following rule.

- If f and g are state formulas, $\mathbf{X}f$ and $f\mathbf{U}g$ are path formulas.

The following abbreviations are also used in writing CTL* and CTL formulas:

- $f \wedge g \equiv \neg(\neg f \vee \neg g)$
- $\mathbf{A}(f) \equiv \neg\mathbf{E}(\neg f)$
- $\mathbf{F}(f) \equiv true\mathbf{U}f$
- $\mathbf{G}(f) \equiv \neg\mathbf{F}\neg f$

◆

The semantics of CTL* are defined with respect to a (finite Kripke) structure $K = \langle W, R, L\rangle$, where

- W is a set of states or worlds.
- $R \subseteq W \times W$ is the transition relation. R is total. $w_1 \rightarrow w_2$ indicates that $(w_1, w_2) \in R$.
- $L : W \rightarrow P(AP)$ is a function that labels each state with a set of atomic propositions true in that state.

Let f_1 and f_2 be state formulas, g_1 and g_2 be path formulas. A *path* in K is defined as a sequence of states $\pi = w_0, w_1, \ldots$ such that for every $i \geq 0, w_i \rightarrow w_{i+1}$. π^i denotes the *suffix* of π starting at w_i. $K, w \models f$ means that f holds at state w in structure K. Similarly, if g is a path formula, $K, \pi \models g$ means that g holds along path π in structure K. The relation $\models$ is inductively defined as follows.

- $w \models A$ IFF $A \in L(w)$.
- $w \models \neg f_1$ IFF it is not the case that $w \models f_1$.
- $w \models f_1 \vee f_2$ IFF $w \models f_1$ *or* $w \models f_2$.
- $w \models \mathbf{E}(g_1)$ IFF there exists a path π starting with w such that $\pi \models g_1$.
- $\pi \models f_1$ IFF w is the first state of π and $w \models f_1$.
- $\pi \models \neg g_1$ IFF it is not the case that $\pi \models g_1$.

- $\pi \models g_1 \vee g_2$ IFF $\pi \models g_1$ *or* $\pi \models g_2$.
- $\pi \models \mathbf{X} g_1$ IFF $\pi^1 \models g_1$.
- $\pi \models g_1 \mathbf{U} g_2$ IFF there exists a $k \geq 0$ such that $\pi^k \models g_2$ and for all $0 \leq j < k$, $\pi^j \models g_1$.

6.2.1.4 Fixed Points

A *functional* is a function that maps functions to functions, that is, a function that takes functions as arguments and returns functions as values. A functional may be denoted by a lambda expression, $\lambda x.f$, where x is a variable and f is a formula. The variable x is effectively a place holder. When the functional is applied to a parameter, p, p is substituted for all instances of x in f.[19] For example, if $\tau = \lambda x.(x \wedge y)$, then $\tau(true) = true \wedge y = y$. A functional γ is *monotonic* if $p \subseteq q \rightarrow \gamma(p) \subseteq \gamma(q)$.

The following definition and example are taken from a discussion in [McM93, p. 19]. A *fixed point* of a functional γ is any p such that $\gamma(p) = p$. For example, if τ is defined as above, then $x \wedge y$ is a fixed point of τ, since $\tau(x \wedge y) = (x \wedge y) \wedge y = x \wedge y$.

A monotonic functional has a *least fixed point* and a *greatest fixed point*, also referred to as *extremal fixed points.* The least (greatest) fixed point was defined by Tarski [Tar55] as the intersection (union) of all the fixed points of the functional. The least and greatest fixed points of a functional $\lambda x.f$ are denoted $\mu x.f$ and $\nu x.f$, respectively. Assuming the functional is continuous, the extremal fixpoints can be characterized as the limit of a series defined by iterating the functional.

The following definitions are also taken from [McM93, p. 19]. A functional, γ, is *union-continuous* (*intersection-continuous*) if the result of applying γ to the union (intersection) of any nondecreasing infinite sequence of sets is equal to the result of taking the union (intersection) of γ applied to each element of the sequence. Tarski showed that if a functional is monotonic and union-continuous, the least fixed point of the functional is the union of the sequence generated by iterating the functional starting with the initial value false, that is, for any such functional, γ, the least fixed point is $\cup_i \gamma^i(false)$. Similarly, the greatest fixed point of a monotonic, intersection-continuous functional, γ, is $\cap_i \gamma^i(true)$.

Any monotonic functional is necessarily continuous (that is, union-continuous and intersection-continuous) over a finite set of states [McM93, p. 19]. Fixed points of functionals have been used to characterize CTL operators, resulting in efficient algorithms for temporal logic model checking. The standard reference for fixed point characterizations of CTL formulas is [EL86].

---

[19] The discussion assumes the usual restrictions on lambda-conversion that ensure that variables occurring free in p are not bound by operators or quantifiers in f.

6.2.1.5 The Mu-Calculus

The mu-calculus is a logic based on extremal fixed points that is obtained by adding a recursion operator, μ, to first-order predicate logic (FOL) or to propositional logic. In the context of FOL, the μ operator can be viewed as an "alternative quantifier for relations" that replaces the standard quantifiers $\forall$ and $\exists$ on relations (but not on individuals) [Par76, p. 174], while in propositional logic, the μ operator provides new n-ary connectives. Kozen [Koz83] credits Scott and De Bakker [SB69] with originating the mu-calculus and Hitchcock and Park [HP73], Park [Par70], and De Bakker and De Roever [BR73] with inititially developing the logic. Park [Par76, p. 173] notes that the mu-calculus was a natural response to the inability of first-order predicate logic "to express interesting assertions about programs" in a reasonable way. The mu-calculus is "strictly intermediate" in expressive power between first- and second-order logics [Par76]. There are several different formulations of the mu-calculus. Some, like those of [BR73, HP73], present the calculus as a polyadic relational system that suppresses individual variables and replaces existential quantification ($\exists$) on individuals with a composition operator on relations [Par76]. Others, like the version below reproduced from [McM93, pp. 114-115] and based on [Par76], retain the more traditional system of predicate logic.

There are two kinds of mu-calculus formulas: *relational* formulas and *Boolean* formulas, and, correspondingly, two kinds of variables: *relational* variables (for example, the transition relation, R) and *individual* variables (for example, the state, x). A model for the mu-calculus is a triple $M = (S, \phi, \psi)$, where S is a set of states, ϕ is the *individual interpretation* function that maps every individual variable to an element of S, and ψ is the *relational interpretation* that maps every n-ary relational variable onto a subset of S^n. The syntax of Boolean formulas is defined as follows, where p and q are syntactic variables representing Boolean formulas, x is an individual variable, $(x_1, \ldots, x_n)$ is a vector of individual variables, and R is an n-ary relational formula.

- *true* and *false* are Boolean formulas.
- $p \vee q$ and $\neg p$ are Boolean formulas.
- $\exists x.p$ is a Boolean formula.
- $R(x_1, \ldots, x_n)$ is a Boolean formula.

The formula $\exists x.p$ is true just in case there exists a state x in S such that p is true in x. Similarly, the formula $R(x, y)$ is true just in case the pair $(\phi(x), \phi(y))$ is a member of $\psi(R)$.

The relational formulas are defined as follows, where, in addition to the definitions given above, F is an n-ary relational formula that is formally monotonic in R.

- Every n-ary relational variable R is an n-ary relational formula.
- $\lambda(x_1, \ldots, x_n).p$ is an n-ary relational formula.
- $\mu R.F$ and $\nu R.F$ are relational formulas.[20]

In a given model (S, ϕ, ψ),

- The relational variable R is identified with the relation $\psi(R)$.
- $\lambda(x_1, \ldots, x_n).p$ denotes the set of all n-tuples $(x_1, \ldots, x_n)$ such that p is true.
- The formulas $\mu R.F$ and $\nu R.F$ stand for the least fixed point and greatest fixed point (of $\tau = \lambda R.F$), respectively.

6.2.2 A Brief History of Finite-State Methods

Finite-state methods grew out of several independent developments in the mid to late 1970s, including early work on temporal logic and early activity in protocol specification and verification. Pnueli first proposed the use of temporal logic to reason about concurrent and reactive programs [Pnu77]. Formalization of safety properties for concurrent systems followed shortly thereafter. Pnueli's early proofs were largely manual, as were the initial techniques used to verify protocols. The realization that many concurrent programs can be viewed as communicating finite-state machines combined with results in reachability analysis and the realization of their applicability to protocol analysis soon led to techniques for automatic verification of correctness properties. [21]

The first such techniques arose in the context of protocol validation [BJ78, Haj78, WZ78, RE80]. Shortly thereafter, in the early 1980s, Sifakis and his students at Grenoble University in France began work on the French validation system Cesar [Que82, QS82], and Harvard colleagues Clarke and Emerson introduced temporal logic model checking algorithms [CE81] that subsequently led to the work by Clarke and his students at Carnegie Mellon University (CMU) on the Extended Model Checker (EMC) system [CES86]. Although Cesar and EMC represent independent developments, both systems used algorithms for the branching-time logic CTL. The CMU system also incorporated slight modifications to CTL to accommodate fairness constraints [BCM$^+$90]. The first general protocol verifier, built by Holzmann and based on reachability analysis, also appeared in the early 1980s [Hol81]. Holzmann's initial protocol verifier employed a simple process algebra, but his subsequent systems use standard automata theory. In all three cases, this early work led to currently important systems: Holzmann's work culminated in SPIN, the Grenoble effort produced Cesar and several specialized variants, and CMU's EMC evolved into SMV.

---

[20] ν may be defined in terms of μ ($\nu R.F[R] = \neg \mu R. \neg F[\neg R]$) or specified as a (primitive) fixpoint operator as shown here.

[21] Initially, safety properties. Liveness and fairness followed later.

Research in model checking for verifying network protocols and sequential circuits quickly led to the realization that application of model checking techniques to nontrivial systems required viable approaches to the so-called *state explosion problem.* The term refers to the fact that in the worst case, the number of states in the global state graph for a system with N processes may grow exponentially with N. There has been a great deal of work on the computational complexity of model checking algorithms, as well as on techniques to address the state explosion problem. One of the earliest and most important techniques for CTL-based model checking systems is a symbolic, rather than an explicit, representation of the state space. That is, the set of states is represented by a logical formula that is satisfied in a given state if and only if the state is a member of the set, rather than by a labeled global state graph. Similarly significant benefits for LTL-based model checking have been obtained with partial order techniques [God90, Val90, Pel93, GPS96]. For certain applications, both techniques can reduce exponential growth of the state space to linear or sublinear growth [Hol].

To provide further economies for CTL-based model checking, symbolic representations capable of exploiting structural regularities and thereby avoiding explicit construction of the state graphs of modeled systems have been sought. The representation that is currently most widely used is a canonical, but highly compact form for Boolean formulas known as *ordered binary decision diagrams* or OBDDs [Bry86].[22] An OBDD is similar to a binary decision tree, except that its structure is a directed acyclic graph rather than a tree and a strict order governs the occurrence of variables. Bryant [Bry86] has shown that there is a unique minimal OBDD for a given formula under a given variable ordering. Variable ordering is thus critical for determining the size of the minimal OBDD for a given formula. Although the use of symbolic representation allows significantly larger systems to be modeled, the state explosion problem persists as a computational barrier restricting the size and complexity of systems that can be verified using finite state methods.

Other strategies have been and continue to be proposed to address this problem. These include exploiting structural symmetries in the systems to be verified [CFJ93, ES93, ID93], using hierarchical [MC85] and compositional [CLM89, GS90] techniques, applying abstraction methods [CGL92, Kur94], and employing on-the-fly intersection techniques [Hol84, CVWY92, FMJJ92]. For LTL-based model checking, efficient on-the-fly techniques have been a significant development because on-the-fly verification algorithms require only that part of the graph structure necessary to prove or disprove a given property, rather than the entire Kripke structure (for example, as required by fixpoint algorithms). Compositionality and abstraction exemplify a "divide-and-conquer" strategy that attempts to reduce the verification problem to a series of potentially more manageable subproblems [God96, p. 17], whereas partial order and on-the-fly methods attempt to reduce the size of the checked state space and the extent of the search, respectively. Some of these techniques may be usefully combined. Partial order and on-the-fly methods are a good example, as noted in [Pel94]. Others are complementary. Compositional and abstraction methods, for example, are essentially orthogonal – and thereby complementary to – partial order techniques [God96, p. 17].

---

[22] OBDD is sometimes written simply as BDD, although as McMillan notes [McM93, p. 32], the variable ordering (which is crucial to obtaining the canonical reduced form) is what distinguishes OBDDs from the more general class of BDDs.

6.2.3 Approaches to Finite-State Verification

As noted earlier, finite-state verification techniques emerged in the late 1970s and early 1980s from two independent developments: temporal logic model checking [CE81,Que82] and protocol analysis [Haj78, Wes78]. Subsequent developments can be classified with respect to several dimensions, reflecting factors such as representation strategy, type of algorithm, and class of system addressed. The distinctions made by representation strategy are broad and therefore well-suited to the general discussion offered here. Representation strategy distinguishes approaches that use a finite state representation for the system model and a logical calculus for the specification—the symbolic model checking approach, from techniques that use finite state machines to represent both the system model and the specification—the automata-theoretic approach. In practice, verification systems for asynchronous systems (software) are largely automata-based, exploit on-the-fly techniques, and support LTL, while systems for synchronous systems (hardware) are based either on fixpoint algorithms or symbolic methods, and support CTL, CTL*, or propositional mu-calculus [Hol].

6.2.3.1 The Symbolic Model Checking Approach

In the symbolic model checking approach, verification means determining whether a given logic formula f is valid in a given Kripke model M, that is, determining which states S in a finite Kripke structure $M = \langle S, R, L \rangle$ satisfy f. Initially, the temporal logics CTL, CTL*, and LTL were used. Later algorithms typically characterize the CTL (LTL) operators (or more precisely, the interpretation of CTL (LTL) operators in a Kripke model) in the Mu-calculus, a logic of extremal fixed points that has been shown to be strictly more expressive than CTL [EL85].[23] The Mu-calculus is attractive because it can be used to express a variety of properties of transition systems and provides a general framework for describing model checking algorithms. A model checking algorithm for the Mu-calculus taken from [BCM^{+}90, p. 7] is presented in Figure 6.1.

Verification systems that perform temporal logic model checking are generally referred to as model checkers, reflecting the fact that the basic function of these systems is to decide whether a given finite model (that is, a Kripke model) satisfies a formula in a given logic. Models are expressed in suitable languages, and assertions about the model are specified in a different language, typically a temporal logic. In the context of

---

[23] A language L' is strictly more expressive than a language L if there are formulas that can be expressed in L' but not in L, and all formulas expressable in L are also expressable in L'.

```
function Bdd_f(f: formula, I_p: rel-interp) : BDD;
   case
     f: an individual variable
        return Bdd_Atom(f);
     f: of the form f1 AND f2
        return Bdd_And(Bdd_f(f1, I_p), Bdd_f(f2, I_p));
     f: of the form NOT f1
        return Bdd_Negate(Bdd_f(f1, I_p));
     f: of the form EXISTS x [f1]
        return Bdd_Exists(x, Bdd_f(f, I_p);
     f: of the form Z(x1,...,xn)
        return Bdd_R(Z, I_p)(d1 <- x1)...(dn <-  xn);
   end case;

function Bdd_R(R: rel-term, I_p: rel-interp) : BDD;
   case
     R: a relational variable
        return I_p(R);
     R: of the form LAMBDA x1,...,xn [f]
        return Bdd_f(f, I_p)(x1 <- d1)...(xn <-  dn);
     R: of the form MU Z [R1]
        return FixedPoint(Z, R1, I_p, FalseBdd);
   end case;

function FixedPoint(Z: rel-var, R: rel-term,
                    I_p: rel-interp, T_i: BDD) : BDD;
   let T_i+1 = Bdd_R, R_p(Z <- T_i);
   if T_i+1 = T_i return T_i
      else return FixedPoint(Z, R, I_p, T_i+1);
```

Figure 6.1: Burch *et al.*'s Mu-Calculus Model Checking Algorithm.

model checking, a suitable language is a reasonably expressive, high-level language, with a precise mathematical semantics that defines its translation to Boolean formulas (OBDDs) or other forms suitable for symbolic model checking.[24] There are several varieties of model checkers, the most common being LTL model checkers that verify linear-time properties of finite Kripke models, and CTL model checkers that verify branching-time properties of finite Kripke models.

---

[24] Although BDDs are still the most widely used symbolic representation for finite state verification, other representations have been used instead of or in addition to BDDs. For example, LUSTRE is a synchronous dataflow language stylistically similar to the SMV language. Verimag's POLKA system (one of several systems to evolve from Cesar) is used to verify LUSTRE [HCRP91, HLR92, HFB93] programs with integer variables. POLKA uses convex polyhedra to represent linear constraints. Recently, a new data structure named Queue-content Decision Diagram (QDD) has been introduced for representing (possibly infinite) sets of queue-contents. QDDs have been used to verify properties of communication protocols modeled by finite-state machines that use unbounded first in, first out (FIFO) queues to exchange messages [BG96]. QDDs have also been used in combination with BDDs to improve the efficiency of (BDD-based) symbolic model-checking techniques [GL96].

For example, the SMV system [McM93,CMCHG96], one of several CMU systems to evolve from EMC, uses a synchronous dataflow language (also called SMV) with high-level operations and nondeterministic choice. The transition behavior of an SMV program, including its initial state(s), is determined by a collection of parallel assignments, possibly involving a unit of delay. Asynchronous systems may be modeled by introducing processes that have arbitrary delay. The SMV language supports modular hierarchical descriptions, reuse of components, and parameterization [CMCHG96, p. 420]. An SMV program consists of a Kripke model and a CTL specification. The state of the model is defined as the collection of the program's state variables, and its transition behavior is determined by the collective effect of the parallel assignment statements. Variables are restricted to finite types, including Boolean, integer subrange, and enumerated types. The SMV program in Figure 6.2 for a very simple protocol illustrates the basic idea. The example is from McMillan [McM93].

```
MODULE main
VAR
  request: boolean;
  state: {ready,busy};
ASSIGN
  init(state) := ready;
  next(state) := case
                   state = ready & request : busy;
                   1 : {ready,busy};
                 esac;
SPEC
  AG(request -> AF state = busy)
```

Figure 6.2: A Simple SMV Program [McM93, p. 63].

Values are chosen nondeterministically for variables that are not assigned a value or whose assigned value is a set. For example, the variable `request` is not assigned in the program, but chosen nondeterministically by the SMV system. Similarly, the value of the variable `state` in the next state is chosen nondeterministically from the values in the set {`ready, busy`}.[25] The specification states that invariantly, if `request` is true, then the value of `state` is `busy`. An SMV program typically consists of reusable modules. SMV processes (not illustrated here) are module instances introduced by the keyword `process`. Safety and liveness properties are expressed as CTL specifications. Fairness is specified by means of *fairness constraints* that restrict the model checker to execution paths along which a given CTL formula is true infinitely often.

---

[25] Like uninterpreted types, nondeterminism can be useful for describing systems abstractly (where values of certain variables are not determined) or at levels that leave design choices open (to the implementor).

6.2.3.2 The Automata-Theoretic Approach

In the automata-theoretic approach, verification means comparing the externally visible behaviors of the finite state machine representing a system model with the finite state machine representing its specification. The method of comparison varies, depending on the technique and the particular class of system for which it was developed.

6.2.3.2.1 Language Containment

In the language intersection approach first described by [VW86], verification consists of testing *inclusion* between two ω-automata, where one automaton represents the system that is being verified and the other represents its specification or *task*. Inclusion denotes the strict subset relation between the languages of the two automata. For a process P modeling a system to be verified and a task T that P is intended to perform, verification consists of the test $\mathcal{L}(P) \subset \mathcal{L}(T)$, where $\mathcal{L}(P)$ denotes the set of all "behaviors" of the modeled system and $\mathcal{L}(T)$ denotes the set of all "behaviors" consistent with the performance of the modeled task or specification. Typically, P is a system of coordinating processes modeled by the product process $P = \otimes P_i$, where each P_i is an ω-automaton.[26] This semantic model accommodates specific *reduction* algorithms that provide one response to the computational complexity problems associated with more general model checking. The basic idea is to replace a computationally expensive test $\mathcal{L}(P) \subset \mathcal{L}(T)$ with a computationally cheaper test $\mathcal{L}(P') \subset \mathcal{L}(T')$, such that $\mathcal{L}(P') \subset \mathcal{L}(T') \Rightarrow \mathcal{L}(P) \subset \mathcal{L}(T)$. P' and T' are derived from P and T, respectively, by homomorphisms on the underlying Boolean algebra.[27]

The reduction of P is *relative* to T, that is, relative to a given task or specification; each task induces a different reduction. Kurshan [Kur94] develops the theory underlying such reductions.

---

[26] For purposes of this discussion, the distinction between finite state machines or generators (of behavior) and finite state automata or acceptors (of behavior) has been glossed over. The former is most convenient for modeling a system and the later for modeling its properties. Interested readers should see [VW86] or [Tho90].

[27] A Boolean algebra is a set closed under the Boolean operations $\wedge$, $\vee$, $\neg$. A homomorphism is a mapping (function) from one algebraic structure to another that is defined in terms of the algebraic operations on the two structures. In the case of two Boolean algebras, B and B', a map ϕ is a homomorphism just in case

$$\phi(x \wedge y) = \phi(x) \wedge \phi(y)$$

$$\phi(x \vee y) = \phi(x) \vee \phi(y)$$

$$\phi(\neg x) = \neg \phi(x)$$

The verification system typically associated with the language-inclusion approach is COSPAN (*C*oordination *Sp*ecification *An*alyzer)[28]. COSPAN's native language is S/R, a data-flow language based on the selection/resolution model of coordinating processes. S/R distinguishes *state* variables from *combinational* variables, the latter being dependent variables whose values are functions or relations of the state variables. The S/R language provides nondeterministic, conditional ("if-then-else") variable assignments; bounded integer, enumeration, Boolean, array, record, and (array and record) pointer types; and integer and bit-vector arithmetic. S/R also supports modular hierarchal development, scoping, parallel and sequential execution, homomorphism declaration, general ω-automaton fairness (acceptance), and (ω-regular) property specification [HHK96, p. 425]. COSPAN provides both symbolic- (that is, BDD-based) and explicit-state enumeration algorithms.

6.2.3.2.2 State Exploration

The terms "state exploration" and "reachability analysis" refer to finite-state verification techniques that begin with an initial state and explicitly enumerate or construct the reachable state space of a system model, typically using standard search algorithms—such as depth-first or breadth-first search—that have been optimized to alleviate state-space explosion. The state exploration approach contrasts with BDD-based techniques, which use a symbolic (implicit) representation of the state space. SPIN [HP96] and Murϕ [Dil96] exemplify this approach. Both verifiers use an asynchronous, *interleaving* model of execution in which atomic operations from a collection of processes execute in an arbitrary order.

SPIN. SPIN is automata-based and has full LTL model-checking capability. Each process of the model is translated into a finite automaton. Properties to be checked are represented as Büchi automata that correspond to a *never* claim, so-called because these claims formalize behavior that should never occur. In other words, never-claims correspond to *violations* of given correctness properties. A model is checked against its required properties by calculating the intersection of the property automaton and the process automata. A nonempty intersection indicates a possible correctness property violation. SPIN uses a verification procedure based on *reachability analysis* of a model by means of optimized graph traversal algorithms. This approach is also referred to as *state exploration*.

The SPIN model checker uses a nondeterministic, guarded command language called PROMELA that was developed to specify and validate protocols by modeling process interaction and coordination. PROMELA provides variables and general control-flow structures in the tradition of Dijkstra's guarded command language [Dij76] and Hoare's language CSP [Hoa85]. Correctness criteria are formalized in PROMELA in terms of assertions that capture both local assertions and global system invariants, labels that can be used to define frequently used correctness claims for both terminating and cyclic sequences (for example, deadlock, bad cycles, and liveness (acceptance and progress) properties), and general temporal claims that define temporal orderings of properties of states expressed either as never-claims or as LTL formulas (that SPIN translates into PROMELA never-claims) [Hol91, HP96].

---

[28] COSPAN is also used as the "verification engine" in the commercial hardware verification tool FormalCheck, a trademark of the Bell Labs Design Automation center [HHK96].

SPIN uses depth-first search and a single-pass, *on-the-fly* verification algorithm coupled with *partial order reduction* techniques to reduce the state explosion problem. On-the-fly algorithms attempt to minimize the amount of stored information, computing the intersection of the process and property automata only to the point necessary to establish the nonemptiness of the resulting (composite) automaton. Partial order reduction algorithms exploit the observation that the order in which concurrent or independently executed events are interleaved typically has no impact on the checked property. It follows that instead of generating all execution sequences, it is sufficient to generate a reduced state space composed of representatives for classes of sequences that are not distinguishable with respect to execution order. The reduction must be shown to preserve safety and liveness properties, but this is accomplished in the course of the verification.

Murϕ. The name "Murϕ" refers both to a verifier developed to analyze finite-state concurrent systems such as protocols and memory models for multiprocessors, and to its language. The Murϕ description language is a guarded-command language based on a Unity-like formalism [CM88] that includes user-defined datatypes, procedures, and parameterized descriptions. A Murϕ description consists of a collection of constant and type declarations, variable declarations, transition rules (guarded commands), start states, and invariants. Predefined data types include subranges, records, and arrays. Murϕ statement types include assignment, condition, case selection, repetition (for- and while-loops), and procedure calls. Murϕ rules consist of a condition and an action. A condition is a Boolean expression on the global variables, and an action is an arbitrarily complex statement. Each rule is executed atomically, that is, without interference from other rules.

Correctness requirements are defined in Murϕ in terms of invariants written as predicates or conditions on the state variables. Invariants are equivalent to `error` statements, which may also be used to detect and report an error, that is, the existence of a sequence of states beginning in a start state and terminating in a state in which a given invariant fails to hold. In addition to invariant violations, error statements, and assertion violations, Murϕ can check for deadlock and, in certain versions, liveness properties.

Murϕ uses standard breadth- or depth-first search algorithms to systematically generate all reachable states, where a state is defined as the current values of the global variables. State reduction techniques, including symmetry reduction, reversible rules, replicated component abstraction, and probabilistic algorithms are exploited to alleviate state explosion [Dil96]. Symmetry reduction uses structural symmetries (in the modeled system) to partition the state space into equivalence classes, thereby significantly reducing the number of states generated in applications such as certain types of cache coherence protocols [ID93]. Reversible rules are rules that preserve information and can therefore be executed "backwards," yielding an optimization that avoids storing transient states [ID96a]. Systems with identical replicated components can be analyzed using explicit state enumeration in an abstract state space in which the exact number of replicated components is treated qualitatively (for example, zero, one, or more than one replicated components) rather than quantitatively (the exact number of replicated components) [ID96b]. The combination of symmetry reduction, reversible rule exploitation, and replicated component abstraction has been reported to yield massive reductions in the state explosion problem for cache coherence protocols and similar applications [Dil96, p. 392]. Probabilistic verification algorithms are being explored as a way of reducing the number of bits in the hash table entry for each state [SD96].

6.2.3.2.3 Bisimulation Equivalence and Prebisimulation Preorders

Bisimulation equivalence provides a logical characterization of when two systems are equivalent and is used to check statewise isomorphism between two finite Kripke models. Prebisimulation preorders similarly provide a logical characterization of when one system minimally satisfies another. Informally, this means that bisimulation provides a notion of behavioral equivalence: two systems are equivalent if they exhibit the same behavior, whereas prebisimulation provides a notion of behavioral relatedness: one system exhibits at least certain behaviors exhibited or required by the other. In both cases, a more abstract or higher-level system serves as a specification of a lower-level one. Verification consists of showing that the lower-level model or "implementation" satisfies its specification by establishing the given relation between the two models. For example, the correctness of a protocol can be established by showing that it is semantically equivalent to its service specification by modeling both the protocol and its specification as finite state machines and using equivalence-checking verification to establish the statewise, transition-preserving correspondence between the two finite-state models. Various formal relationships have been proposed. In general, these relations are either equivalences (bisimulations) or preorders (prebisimulations) [CH89].

Milner's Calculus of Communicating Systems (CCS[29]) [Mil89] forms the basis for several of the most visible equivalence-checking verifiers for concurrent systems. Processes are defined as CCS agents that are given an operational semantics defined in terms of transition relations. CCS processes may define an arbitrary number of subprocesses, in which case the transition graph may have infinitely many states. Although some properties may be decidable in such cases, most interesting properties are undecidable on agents that correspond to graphs with infinite state spaces. Automated tools for analyzing networks of finite-state processes defined in CCS include the NCSU Concurrency Workbench [CS96] and its predecessor, the (Edinburgh) Concurrency Workbench [CPS93], and the Concurrency Factory [CLSS96]. Both versions of the Concurrency Workbench support equivalence checking, preorder checking, and model checking (for the modal mu-calculus). The NCSU Concurrency Workbench also provides diagnostic information if two systems fail to be related by either semantic equivalence of preorder, and language flexibility that allows the user to change the system description language [CS96]. The Concurrency Factory is also an integrated toolset, but focuses on practical support for formal design analysis of real-time current systems. This is achieved in part through a graphical design language (GCCS), a graphical editor, and a graphical simulator [CLSS96]. In addition to a CCS-based semantics, GCCS has a structural operational semantics [CLSS96, p. 400].

---

[29] CCS and related approaches are also referred to as *process algebras*.

6.2.4 Utility of Finite-State Methods

The various approaches to finite-state verification outlined earlier are in theory very similar and in many cases inter-definable, as noted in [VW86,CGK89,CBK90]. In practice, the approaches have led to the development of tools with often overlapping capabilities, but different foci and strategies. For example, SPIN has been developed for modeling distributed software using an asynchronous process model; Murϕ and SMV have focused on hardware verification—Murϕ on asynchronous concurrent systems using explicit state exploration and SMV on both synchronous and asynchronous systems using symbolic model checking; and COSPAN has been driven by a top-down design methodology implemented through successive refinement of (fundamentally) synchronous models and has been used for both software and hardware design verification. In some cases, the capabilities are complementary, and there is work on integrating different finite-state verification strategies as done in COSPAN, which offers either symbolic- (BDD-based) or explicit state enumeration algorithms, as well as on integrating different approaches in a single tool, as done in both versions of the Concurrency Workbench, which offer equivalence checking, preorder checking, and model checking.

Finite-state methods offer powerful, automated procedures for checking temporal properties of finite-state and certain infinite-state systems (Kripke models). They also have the ability to generate *counterexamples*—typically in the form of a computation path that establishes, for example, the failure of a property to hold in all states, and *witnesses*—in the form of a computation path that establishes the existence of one or more states in which a property is satisfied. Finite-state methods are least effective on large, unbounded state spaces, high-level specifications, and data-oriented applications—areas in which deductive methods are more appropriate. For this reason, there has been increasing interest in integrating finite-state methods and deductive theorem proving. This topic is revisited in Section 6.4.

6.3 Direct Execution, Simulation, and Animation

Direct execution, simulation, and animation are techniques used to observe the behavior of a model of a system. Formal analysis, on the other hand, is used to analyze modeled behavior and properties. In many cases, there are fundamental differences between these observational and analytical methods, including the models they use and their expected performance. Typically, models used for verification cannot expose their own inaccuracy and, conversely, models used for conventional simulation cannot confirm their own correctness [Lan96, p. 309]. Models used for simulation of large systems must be able to handle realistic test suites fast, since these suites may literally run for weeks. This kind of efficiency is not a reasonable expectation in executable specification languages. Formal verification techniques generally treat the notion of time as an abstraction and largely avoid probabilities, whereas more concrete representations of time and probabilistic analyses play an important role in observational methods. Finally, direct execution, simulation, and animation show behavior over a finite number of cases, whereas formal analysis can be used to explore all possibilities, the former offering statistical certainty and the latter, mathematical certainty. Although some of these differences are attenuated when "simulation" is considered in the context of formal specification languages (for example, the models used for execution and simulation typically coincide), others persist (for example, verification still proceeds by extrapolation from a finite number of cases, rather than by mathematical argumentation over all possible cases). The remainder of this section summarizes the notions of executability, simulation, and animation in the context of formal methods.[30]

6.3.1 Observational Techniques

Some formal specification languages are directly executable, or contain a directly executable subset, meaning that the specification itself can be executed or *run* and its behavior observed directly. For example, a logic based on recursive functions, such as that used in Nqthm [BM88] and ACL2 [KM94], supports direct execution and "simulation" on concrete test cases because it is always possible to compute the value of a variable-free term or formula in the executable subset of these logics. The following quote from [KM94, p. 8] describes the role of executability in the formalization of a model of a digital circuit (the FM9001) in Nqthm.

> [The Nqthm model] can be thought of as a logic simulator (without, however, the graphic and debugging facilities of commercial simulators). ...Running [the model] on a concrete netlist[31] and data involves simulating in the proper sequence the input/output behavior of every logical gate in the design ...

The specification language for the Vienna Development Method (VDM), VDM-SL, also has a large executable subset, as well as tool support for dynamically checking type invariants and pre and post conditions, and for running test suites against a VDM-SL specification [VDM]. Similarly, the concrete representation of algorithms and data structures required by most finite-state enumeration and model-checking methods (see

---

[30] Planning and administrative trade-offs involving, for example, cost, available resources, criticality of the system, and desired levels of formality, are discussed in the first volume of the guidebook [NASA-95a].

[31] The "netlist" is an Nqthm constant that describes a tree of hardware modules and their interconnections via named input/output lines.

Section 6.2) make them comparable to direct execution techniques. Certain finite state verification tools also provide "simulation," by exploring a single path through the state space rather than all possible paths [Hol91, DDHY92, ID93].

The dynamic behavior of specifications written in nonexecutable languages may be studied indirectly, by reinterpreting the specification in a (high-level) programming language. Execution of the resulting program is referred to as an *emulation* or *animation* of the specification. Some formal specification languages offer both a directly executable subset and the option of user- or system-defined program text to drive animation of nonexecutable parts of the specification. Specifications written in a nonexecutable language using a constructive functional style may be "executed" by exploiting a rewrite facility (assuming one is available) to rewrite function definitions, starting from a particular set of arguments. This amounts to writing an emulator for the system being modeled and may not be either possible or desirable. For example, making an entire specification executable typically precludes using axioms to dispense with those parts of a system or its environment that are not of interest or do not warrant verification.

Direct execution, simulation, and animation are not alternatives to more rigorous formal analysis, but rather effective complements. For example, during the requirements and(or) high-level design phase, executability can be used to probe the behavior of a system on selected test cases, and deductive theorem proving can be used to exhaustively establish its general high-level properties. In this type of strategy, executability provides an efficient way to avoid premature proof efforts and, conversely, to focus the more rigorous (and thereby more expensive) proof techniques on the most appropriate behaviors and properties. This symbiotic use of different techniques is nicely illustrated in the development of a formal specification of the Synergy File System using ACL2 [BC95a]. In this application, formalization of an ACL2 executable model, execution of the model, and proof of an invariant about transitions in the model each revealed significant errors.

6.3.2 Utility of Observational Techniques

The main advantages of executability are that it allows the specification and underlying model to be "debugged," and it allows the specification to serve as a "test oracle" relatively early in the life cycle. Animation and emulation confer similar benefits. A further advantage of executability is that it allows behavior to be observed and explored in the same formally rigorous context as that in which the specification is developed. Other documented roles for executability include post-implementation testing, as illustrated, for example, in post-fabrication execution of the FM9001 specification to test the fabricated devices for conformance to the (verified) design [KM94, p. 9]. Although this example represents a somewhat novel use of executability, it is potentially an important technique by means of which formal methods can make a unique contribution to conventional testing regimes. The technology transfer potential of executability, animation, and emulation is also worth noting. Because simulation, animation, and emulation are techniques familiar to analysts and engineers, they offer an effective vehicle for integrating formal methods into ongoing system development activities. The VDM-

SL study carried out at British Aerospace provides an interesting example of the role of executability in the integration of formal specification in a traditional development process [LFB96].

6.4 Integrating Automated Analysis Methods

No single technique is effective across a wide range of applications or even across a single application with disparate components or algorithms. Industrial-strength examples typically require a variety of approaches, currently used as standalone systems, as illustrated, for example, in [MPJ94]. Rushby [Rus96] argues that effective deductive support for formal methods requires not standalone, but integrated techniques effective across a broad range of applications. Shankar [Sha96] makes a similar argument, noting that the "sheer scale" of mathematics necessary for formal methods argues for a unification of verification techniques.

The three analysis techniques surveyed in this chapter—automated deductive methods, finite-state methods, and simulation methods—have complementary strengths and there is increasing interest in the synergistic integration of these techniques within a uniform framework. Synergistic integration simply means that the resulting system should be more than the sum of its parts. Logical frameworks, such as Isabelle [Pau88], support the definition and construction of deductive tools for specialized logics, but do not provide systematic support for coherent integration of different capabilities [Sha96]. The Stanford TEmporal Prover (STEP) [Man94], which integrates model checking with algorithmic deductive methods (decision procedures) and interactive deductive methods (theorem proving) to support verification of reactive systems, is an example of one strategy in the search for effective integration. The STEP system is interesting because it also combines powerful algorithmic and heuristic techniques to automatically generate invariants. A different approach has been used to integrate model checking and automated proof checking in PVS [RSS95], where a BDD-based model checker for the propositional mu-calculus is integrated as an additional decision procedure within the proof checker.

The notion of integrated verification techniques introduced here provides a glimpse of the direction verification technology is heading. One implication of this discussion is the relative maturity of existing formal methods techniques, which offer effective specification and analysis options for aerospace applications.

6.5 Proof of Selected SAFER Property

The property that no more than four thrusters may be fired simultaneously follows directly from the detailed functional requirements of the SAFER system. Thruster selection is a function of the integrated hand grip and AAH-generated commands. The thruster select logic specified in Tables C.2 and C.3 is used to choose appropriate thrusters based on a given integrated command. An initial survey of these tables might suggest that as many as four thrusters can be selected from each table, resulting in as many as eight thrusters chosen in all. However, several additional constraints render certain command combinations invalid. Furthermore, the table entries themselves are interrelated in ways that limit the thruster count for multiple commands. The four-thruster maximum follows directly from the combination of these two types of constraint.

The four-thruster max property is fundamental and is explicitly captured as Requirement 41, one of the avionics software requirements (see Sections 3.3 and C.2):

41. The avionics software shall provide accelerations with a maximum of four simultaneous thruster firing commands.

The four-thruster max property can be expressed as a PVS theorem as shown here.

```
    max_thrusters: THEOREM
    FORALL (a_in: avionics_inputs), (a_st: avionics_state):
           length(prop_actuators(output(SAFER_control(a_in, a_st)))) <= 4
```

The theorem asserts that for any input and state values, the outputs produced by the SAFER controller, which include the list of thrusters to fire in the current frame, obey the maximum thruster requirement. This claim applies to any output that can be generated by the model.

6.5.1 The PVS Theory `SAFER_properties`

Proof of the `max_thrusters` theorem requires several supporting lemmas. These lemmas and the theorem itself are packaged as the PVS theory `SAFER_properties`, which is reproduced here.

```
SAFER_properties: THEORY
BEGIN

IMPORTING avionics_model

A,B,C:    VAR axis_command
tr:       VAR tran_command
HCM,cmd:  VAR six_dof_command
AAH:      VAR rot_command
state:    VAR AAH_state
thr,U,V:  VAR thruster_list
act:      VAR actuator_commands
BF,LRUD:  VAR thruster_list_pair


%%  Only one translation command can be accepted for thruster selection.

only_one_tran(tr): bool =
        (tr(X) /= ZERO IMPLIES tr(Y) = ZERO AND tr(Z) = ZERO)
           AND (tr(Y) /= ZERO IMPLIES tr(Z) = ZERO)

only_one_tran_pri: LEMMA
        only_one_tran(prioritized_tran_cmd(tr))

only_one_tran_int: LEMMA
        only_one_tran(tran(integrated_commands(HCM, AAH, state)))


%%  All categories of selected thrusters (BF vs. LRUD and mandatory
%%  vs. optional) are bounded in size by two, which follows directly
%%  from inspection of the tables.

max_thrusters_BF: LEMMA
        length(proj_1(BF_thrusters(A, B, C))) <= 2 AND
        length(proj_2(BF_thrusters(A, B, C))) <= 2

max_thrusters_LRUD: LEMMA
        length(proj_1(LRUD_thrusters(A, B, C))) <= 2 AND
        length(proj_2(LRUD_thrusters(A, B, C))) <= 2
```

```
%%  Absence of translation commands implies no optional thrusters
%%  will be selected.

no_opt_thr_BF: LEMMA
        tr(X) = ZERO IMPLIES length(proj_2(BF_thrusters(tr(X), B, C))) = 0

no_opt_thr_LRUD: LEMMA
        tr(Y) = ZERO AND tr(Z) = ZERO IMPLIES
          length(proj_2(LRUD_thrusters(tr(Y), tr(Z), C))) = 0


%%  Top level theorems establishing bounds on number of selected thrusters:

max_thrusters_sel: LEMMA
        only_one_tran(tran(cmd)) IMPLIES
          length(selected_thrusters(cmd)) <= 4

max_thrusters: THEOREM
        FORALL (a_in: avionics_inputs), (a_st: avionics_state):
          length(prop_actuators(output(SAFER_control(a_in, a_st)))) <= 4

END SAFER_properties
```

The `SAFER_properties` theory depends on other theories in the SAFER specification, as shown in the graph of the dependency hierarchy in Figure 6.3. Only the dependency on the theory `avionics_model` is explicitly represented (in the `IMPORTING clause` in `SAFER_properties`). The remaining dependency chains are established through similar clauses in the other theories.

The lemmas in `SAFER_properties` differ in import. Some are used to decompose the proof. Others express general properties of the problem domain that are likely to be useful in the proof of additional SAFER properties as well as in the proof of `max_thrusters`. Annotations (indicated by the PVS comment character `%`) indicate whether the lemma represents an intermediate proof step or a general property.

The mechanically assisted proof of the `SAFER_properties` theory consists of a proof of the top-level theorem, `max_thrusters`, whose proof follows from the lemmas `max_thrusters_sel` and `only_one_tran_int`. Each of these lemmas is, in turn, proved in terms of other lemmas from this theory. The PVS theorem prover employs a sequent calculus similar to that sketched in Section 6.1.3.1.2, but mechanized at a considerably higher level than that reflected in the proof in Section 6.1.3.3. Section C.4.2.2 shows a transcript from the proof of theorem `max_thrusters`. The proof contains only five steps in the PVS theorem prover. Proofs of the remaining lemmas are similarly straightforward and require only a few steps. The single exception is `max_thrusters_sel`, whose proof involves a case analysis.

6.5.2 Informal Argument for Lemma `max_thrusters_sel`

Consider, first, an informal argument for the `max_thrusters_sel` lemma. At most two mandatory and two optional thrusters can be selected from each of the two thruster tables. The argument proceeds by cases defined in terms of possible commands.

The first case concerns a translation command for the X axis.

- **Case 1: No X command present.** Inspection of Table C.2 shows that there will be no optional thrusters selected in this case. There are two subcases, depending on the presence of a pitch or yaw command.
  - **Case 1.1: No pitch or yaw commands.** Inspection of Table C.2 shows that no thrusters at all are selected in this case. At most four can come from Table C.3. Hence, the max thruster property holds.
  - **Case 1.2: Pitch or yaw command present.** Inspection of Table C.3 indicates that no optional thrusters are chosen from this table. Hence, only mandatory thrusters from each table are chosen, and, again, the number selected cannot exceed 4.
- **Case 2: X command present.** Because only one translation command is allowed, it follows that no Y or Z command can appear. This, in turn, implies that no optional thrusters are chosen from Table C.3. The subcases take into account the possibility of a roll command.
  - **Case 2.1: No roll command.** Without a roll command, no thrusters are selected from Table C.3. Hence, the max thruster property holds.
  - **Case 2.2: Roll command present.** A roll command implies that Table C.2 yields no optional thrusters. This leaves only mandatory thrusters from each table, and the bound of four thrusters is satisfied.

The case analysis sketched in this informal proof can be directly formalized in PVS. The resulting proof is quite lengthy, as shown in the proof tree in Figure 6.4. As noted earlier, the level of automation represented in this figure is higher than that illustrated in Section 6.1.3.3.

Although it is certainly possible to use mechanized proof tools to verify informal proofs in this way, it is often far more productive to exploit the strengths of a particular tool to make the proof more automatic, more comprehensible, or more optimal with respect to other desired metrics. This kind of optimization follows quite naturally as one of the later steps in the inherently iterative process of developing and refining a proof. Figure 6.5 shows a considerably simpler and more automated proof for the `max_thrusters_sel` property. This second proof exploits the high-level PVS `GRIND` command that packages many lower-level commands, thereby automating most of the proof of `max_thrusters_sel`.

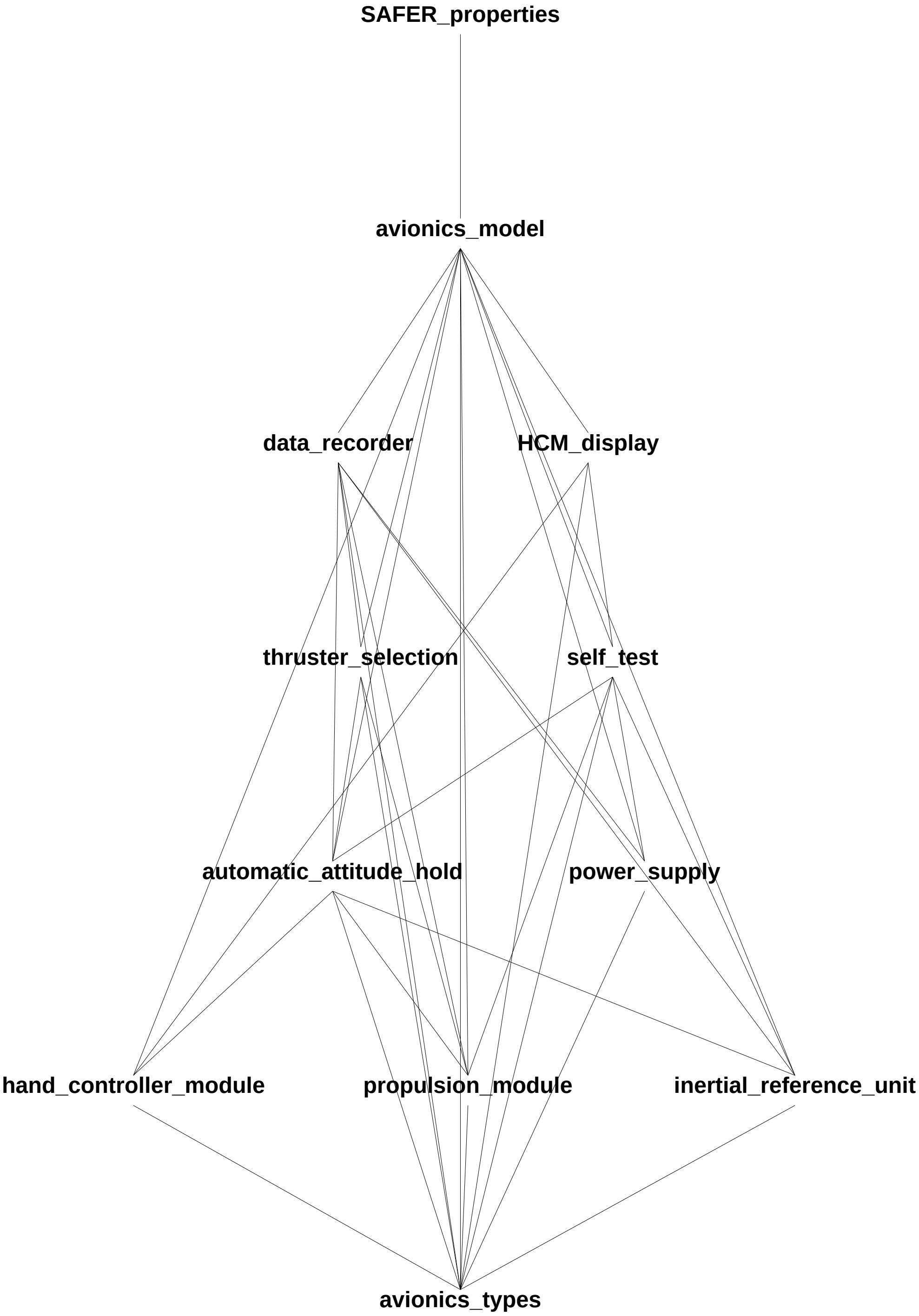

SAFER_properties
avionics_model
data_recorder
HCM_display
thruster_selection
self_test
automatic_attitude_hold
power_supply
hand_controller_module
propulsion_module
inertial_reference_unit
avionics_types

Figure 6.3: Dependency Hierarchy for `SAFER_properties`.

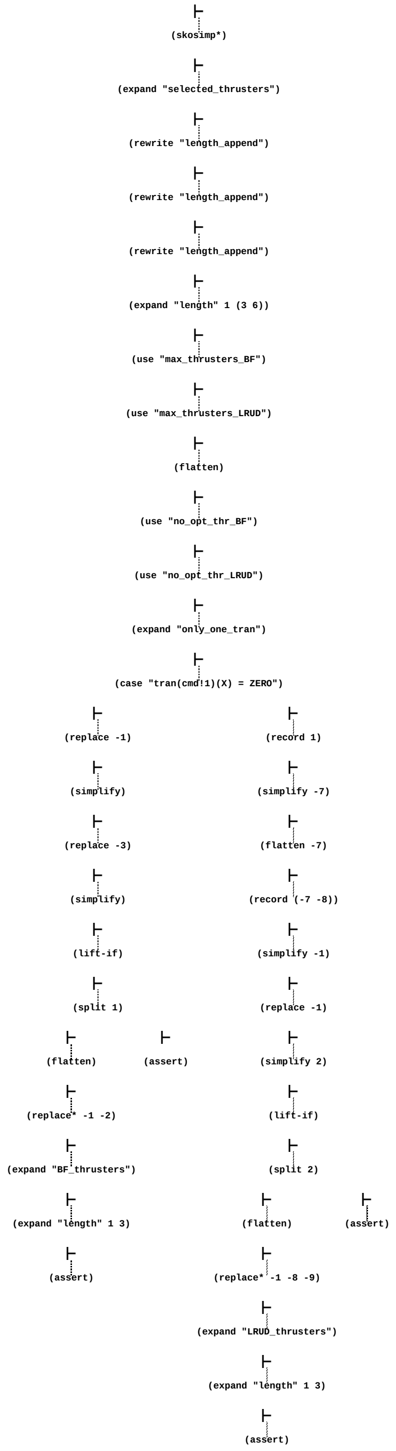

(skosimp*)
(expand "selected_thrusters")
(rewrite "length_append")
(rewrite "length_append")
(rewrite "length_append")
(expand "length" 1 (3 6))
(use "max_thrusters_BF")
(use "max_thrusters_LRUD")
(flatten)
(use "no_opt_thr_BF")
(use "no_opt_thr_LRUD")
(expand "only_one_tran")
(case "tran(cmd!1)(X) = ZERO")
(replace -1)
(record 1)
(simplify)
(simplify -7)
(replace -3)
(flatten -7)
(simplify)
(record (-7 -8))
(lift-if)
(simplify -1)
(split 1)
(replace -1)
(flatten)
(assert)
(simplify 2)
(replace* -1 -2)
(lift-if)
(expand "BF_thrusters")
(split 2)
(expand "length" 1 3)
(flatten)
(assert)
(assert)
(replace* -1 -8 -9)
(expand "LRUD_thrusters")
(expand "length" 1 3)
(assert)

Figure 6.4: Proof Tree for `SAFER_properties_max_thrusters_sel`.

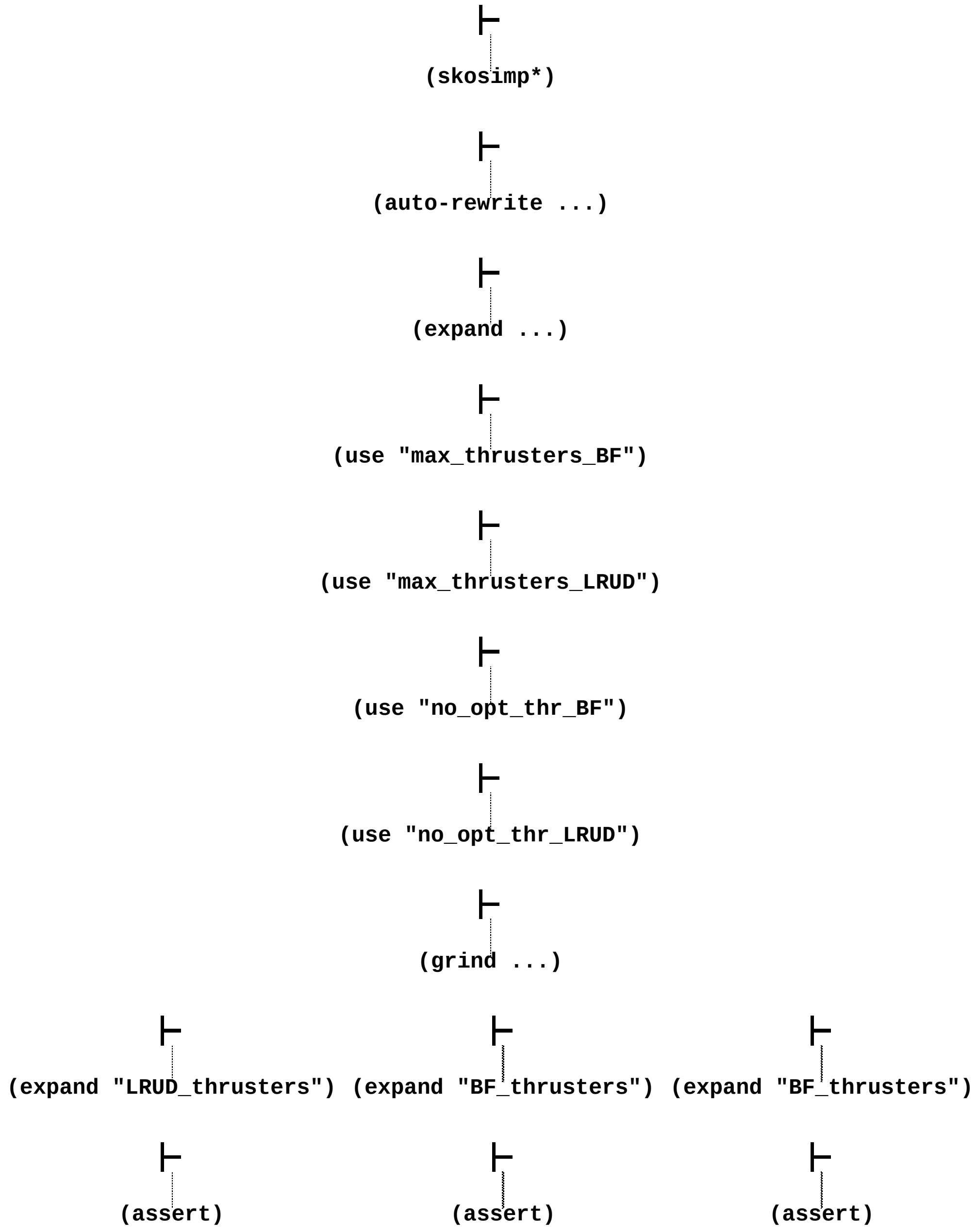

⊢
(skosimp*)
⊢
(auto-rewrite ...)
⊢
(expand ...)
⊢
(use "max_thrusters_BF")
⊢
(use "max_thrusters_LRUD")
⊢
(use "no_opt_thr_BF")
⊢
(use "no_opt_thr_LRUD")
⊢
(grind ...)
⊢
(expand "LRUD_thrusters")
⊢
(assert)
⊢
(expand "BF_thrusters")
⊢
(assert)
⊢
(expand "BF_thrusters")
⊢
(assert)

Figure 6.5: Revised Proof Tree for `SAFER_properties_max_thrusters_sel`.

Chapter 7

Conclusion

This guidebook has presented a discussion of the technical issues involved in the use of formal methods. The focus has been on using formal methods to analyze requirements and high-level designs, that is, on a spectrum of activities that apply mathematical techniques to formalize, explore, debug, validate, and verify software and hardware systems. The development of the SAFER specification has exemplified the process of applying formal methods to aerospace applications.

The guidebook characterizes formal methods as an iterative process whose broad outlines are determined by contextual factors. Effective use of this process involves judiciously pairing formal methods with an application and its careful integration with existing quality control and assurance activities.

7.1 Factors Influencing the Use of Formal Methods

Two types of factors influence the use of formal methods: administrative factors and technical factors. Administrative factors—including project scale and staffing, integration of formal methods with traditional processes, and general project guidelines: training, specification and documentation standards and conventions, and so on—are discussed in Volume I of this Guidebook [NASA-95a]. Technical factors—including the type, size, and structure of the application; level of formalization; scope of formal methods use; characteristics of available documentation, and choice of formal methods tool—have been the subject of this second volume of the guidebook. These technical factors are summarized here.

- **Type, Size, and Structure of the Application** Formal methods are best suited to the analysis of complex problems, taken singly or in combination, and less suited for numerical algorithms or highly computational applications. Applications of moderate size with a coherent structure that can be decomposed into subsystems or components are typically most appropriate.

- **Level of Formalization** Formal methods can be productively applied at various levels of formality or rigor, ranging from the occasional use of mathematical notation to exclusive use of semantically well-defined specification languages with mechanized proof support.
- **Scope of Formal Methods Use** Formal methods can be effectively applied in a variety of ways depending on which stages of the developmental life cycle, which system components, and what system functionality are formalized.
- **Documentation** Formal methods benefit from the availability of adequate documentation. The most important characteristics are the level at which the requirements (high-level design) are stated, the degree to which they are explicitly and unambiguously enumerated, the extent to which they can be traced to specific system components, and the availability of additional information or expertise to motivate and clarify their definition.
- **Tool(s)** Formal methods typically involve some level of mechanical support. The choice of formal methods tool, if any, is determined by administrative factors and the preceding technical factors (excepting documentation). Information on formal methods tools is available from several databases, including those maintained by Jonathon Bowen, Larry Paulson, and Carolyn Talcott, respectively [Bowen,Pauls, Talco].

7.2 The Process of Formal Methods

Contextual factors determine the broad outlines of formal methods use for a given application. The substance of the formal methods process has been characterized in previous chapters of this volume as a discipline composed of the following activities: characterizing, modeling, specifying, analyzing, documenting, and maintaining/generalizing.

- **Characterizing** Synthesizing a thorough understanding of the application and the application domain, resulting in a working characterization of the application and relevant parts of its environment.
- **Modeling** Selecting a mathematical representation expressive enough to formalize the application domain, while providing sufficient analytical power to explore, calculate, and predict the behavior of the system.
- **Specifying** Developing a specification strategy, formalizing the application in terms of the underlying model and articulated strategy, and checking the syntactic and semantic correctness of the specification.
- **Analyzing** Predicting and calculating system behavior, challenging underlying assumptions, validating key properties and invariants, establishing the consistency of axioms, and establishing the correctness of hierarchical layers.

- **Documenting** Recording underlying assumptions, motivating critical decisions, documenting rationale and crucial insights, providing additional explanatory material, tracing specification to requirements (high-level design), tracking level of effort, and collecting cost/benefit data.
- **Maintaining/Generalizing** Revisiting and, as necessary, modifying the specification and analysis to predict the consequences of proposed changes to the modeled system, to reflect mandated changes to the modeled system, to accommodate reuse of the formal specification and analysis, or to distill general principles from the formalization.

Although this linearization of the process is informative, it is important to keep two additional facts in mind. First, applying formal methods is an iterative process. A specification, like a conventional program, must be methodically developed, explored, modified, and refined through many iterations until the result is free of syntactic and semantic errors and captures desired characteristics and behaviors in a concise and easily communicated form. Second, the list is not prescriptive. Each project necessarily selects the most appropriate subset of the activities listed above, namely those most consistent with its mandate and the resources at its disposal.

7.3 Pairing Formal Methods, Strategy, and Task

Formal methods offer a diverse set of techniques appropriate for a wide variety of applications. Moreover, there are many ways to use these techniques to model systems and to calculate and explore their properties. The implications of this rich repertoire of techniques and strategies is that the effective use of formal methods involves judicious pairing of method, strategy, and task. For example, control-intensive algorithms for small finite systems, such as mode sequencing algorithms, are often most effectively analyzed using state exploration, while general properties of complex algorithms, such as Byzantine fault-tolerant clock synchronization, typically require efficient deductive support for arithmetic in the form of arithmetic decision procedures. When an optimal pairing of methods, strategy, and task is not readily apparent, a rapid prototype of an aggressively downscaled or abstracted model that preserves essential properties of interest can help to focus the selection. Precedence, that is, techniques or strategies successfully applied to similar tasks, can also serve as a guide in these cases.

A complex application is typically decomposable into subtasks. In such cases, it may be productive to apply a combination of methods, or to apply a "lightweight" method such as model checking, animation, or direct execution to specific or reduced cases of all or part of a specification before attempting a more rigorous and costly analysis. For example, [HS96] reports the analysis of a communications protocol using a combination of finite state exploration, theorem proving, and model checking. The protocol was first manually reduced to finite state to allow certain safety properties to be checked using finite state exploration. These properties were then verified for the full protocol using deductive theorem proving. The invariant used for the proof had to be strengthened through additional conjuncts discovered incrementally during the proof process. Each proposed new conjunct was checked in the reduced model, using state exploration before it was used in the evolving proof. This iterative process eventually yielded an invariant composed of 57 conjuncts. Exploiting the knowledge gained in this exercise, a finite-state abstraction of the original protocol was developed and mechanically verified. Finally, properties of the abstraction were verified, using a model checker for the propositional mu-calculus (see Chapter 6, Section 6.2.1.5). Although this particular example reflects a demanding exercise carried out by expert practitioners, it is a nice illustration of the productive interaction of combinations of techniques and strategies that are available to expert and nonexpert alike.

7.4 Formal Methods and Existing Quality Control and Assurance Activities

Formal methods complement, but do not replace, testing and other traditional quality control and assurance activities.[1] This symbiotic relationship between formal methods and traditional quality control and assurance methods derives from the fact that formal methods are most effectively used early in the life cycle, on suitably abstract representations of traditionally hard problems,[2] in order to provide complete exploration of a model of possible behaviors. Conversely, traditional quality control and assurance methods have proven highly effective late in the life cycle on concrete (implemented) solutions to hard problems, in order to establish the correctness of detailed and extensive, but necessarily finite behavioral scenarios.

There are many ways to exploit the complementarity between formal methods and existing quality control and assurance activities. Some of these directly target formal methods' products. For example, [CRS96,SH94] describe a fully automatable structural ("black box") specification-based testing technique that complements implementation-based testing. This technique derives descriptions of *test conditions* from a formal specification written in a predicate logic-based language. The test conditions guide selection of test cases and measure the comprehensiveness of existing test suites. Recent conference proceedings, for example [COMP95,ISSTA96], attest to current interest in developing automated methods that use formal specifications to generate test artifacts for concrete implementations.

Other approaches reflect a more indirect use of formal methods. For example, formal, or even quasi-formal models developed during the application of formal methods can be used to facilitate traditional safety analyses. Leveson et al. report [MLR$^+$96, p. 14] that

---

[1] Following Rushby [Rus93b, p. 144], *quality control* denotes "methods for eliminating faults" and *quality assurance* denotes "methods for demonstrating that no faults remain."

[2] Including, but not limited to, fault tolerance, concurrency, and nondeterminism, where capabilities distributed across components must be synchronized and coordinated, and where subtle interactions, for example, due to timing and fault status, must be anticipated.

"...the state abstraction and organization [of their state-transition models] facilitated ...fault tree analysis." A further input to traditional safety analyses might involve the formal specification and analysis of key safety properties. For example, it can be demonstrated that a particular formal model satisfies (or fails to satisfy) given safety properties, that proposed system modifications captured in a model fail to preserve desired safety properties, or that an executable specification fails to satisfy a given test suite. The results of these and other formal analyses can, in turn, be used to expose areas of potential concern and, thereby, concentrate conventional testing activities. If the results of the testing are then iterated back into the formal analysis, the increasingly focused iteration can be used to refine requirements or high-level designs. The examples cited here are suggestive, only. In general, the tighter the integration of formal and conventional methods, the more productive the interplay between formal techniques and traditional quality control and assurance activities.

7.5 Formal Methods: Verification Versus Validation and Exploration

The real value of formal methods lies not in their ability to eliminate doubt, but in their capacity to focus and circumscribe it.[3]

The use of formal methods is often seen as a form of absolute guarantee—a proof of total correctness. However, as Rushby [Rus93b, pp. 74-75] notes, equating formal verification with total correctness is doubly misleading in that it overestimates the guarantee conferred by formal verification while it underestimates the value of the formal verification process, per se.

The guarantee conferred by formal verification assures the mutual consistency of the specifications at either end of a chain of verification, but necessarily fails to address the adequacy of the underlying model, the extent to which the highest-level specification captures the requirements, or the fidelity with which the lowest-level specification captures the behavior of the actual system. The potentially contentious issue of the adequacy of the model is typically resolved through extensive use, challenge, and review, although there have been a few interesting attempts to characterize and automate the selection of "adequate" models of physical systems [Nay95]. The fidelity of the upper- and lowermost specifications in a chain of verification is established through validation.

The value of the process of formal verification lies not only, or even primarily, in the end product—that is, in a proof of correctness, but rather in the benefits accumulated along the way. These benefits include many of those discussed in previous chapters of this guidebook.

- A detailed enumeration of all the assumptions, axioms, and definitions that provide the underlying basis for the verification and characterize the requirements and properties whose satisfaction or utility in the physical world must be empirically validated.

---

[3] Paraphrase of a comment from John Rushby.

- The validation of these assumptions and properties (for example, through proof checking or model checking).
- The (early) detection of inconsistent requirements or of design faults. Most verifications fail, at least initially, and the information gained from these failed attempts reveals unstated assumptions, missing cases, and other errors of interpretation or omission. Although some of these errors would probably be caught by conventional techniques, others are quite subtle and less likely to be exposed by informal techniques or sampled behaviors.
- The ability to explore readily and reliably the consequences of additional or modified assumptions, requirements, and designs, reinforcing and informing the necessarily iterative process of developing large and complex systems.
- The ability to identify and develop reusable formal methods techniques, strategies, and products, contributing to a cost-effective approach to the development of large and complex systems.
- The improved understanding and identification of better solutions derived from the intense scrutiny and discipline involved in the process of formalization and formal analysis.

In summary, formal methods do not focus exclusively or even primarily on "proving correctness"—the verification activities associated with software implementations and hardware layouts—but rather on exploring, debugging, and validating artifacts, such as requirements and high-level designs, leading to a deeper understanding of their properties and assumptions, an earlier capability for calculating and predicting their behavior, and a fuller appreciation of the consequences of modifying their structure, properties, or environment. This guidebook has attempted to provide formal methods practitioners with the information and insight essential to the productive use of formal methods.

References

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, New York, NY, 1996.

[ACHH93] R. Alur, C. Courcoubetis, T. A. Henzinger, and Pei-Hsin Ho. Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems. In Grossman et al. [GNRR93], pages 209—229.

[Ack62] R. L. Ackoff, editor. *Scientific Method: Optimizing Applied Research Decisions*. John Wiley and Sons, 1962.

[ACM94] *Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '94)*, Portland, Oregon, October 1994. ACM/SIGPLAN(in SIGPLAN Notices Vol 29, No. 10, October 1994).

[AD91] R. Alur and D. Dill. The Theory of Timed Automata. In de Bakker et al. [dBHdRR91], pages 45—71.

[AH91] R. Alur and T. A. Henzinger. Logics and Models of Real Time: A Survey. In de Bakker et al. [dBHdRR91], pages 74—106.

[AH95] R. Alur and Pei-Hsin Ho. HYTECH: The Cornell HYbrid TECHnology Tool. In Antsaklis et al. [AKNS95], pages 265—293.

[AH96] Rajeev Alur and Thomas A. Henzinger, editors. *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, New Jersey, July/August 1996. Springer-Verlag.

[AH97] M. Archer and C. Heitmeyer. Verifying Hybrid Systems Modeled as Timed Automata: A Case Study. In *Proceedings of the International Workshop on Hybrid and Real-Time Systems (HART'97)*, Grenoble, France, March 1997.

[AHS96] Rajeev Alur, Thomas Henzinger, and Eduardo Sontag, editors. *Hybrid Systems III, Verification and Control*, volume 1066 of *Lecture Notes in Computer Science*, New York, NY, 1996. Springer-Verlag.

[AINP88] Peter B. Andrews, Sunil Issar, Daniel Nesmith, and Frank Pfenning. The TPS Theorem Proving System. In Lusk and Overbeek [LO88], pages 760–761.

[AKNS95] Panos Antsaklis, Wolf Kohn, Anil Nerode, and Shankar Sastry, editors. *Hybrid Systems II*, volume 999 of *Lecture Notes in Computer Science*, New York, NY, 1995. Springer-Verlag.

[AL95] Y. Ampo and R. Lutz. Evaluation of Software Safety Analysis Using Formal Methods. In *Foundation of Software Engineering Workshop*, Hamana-ko, Japan, December 1995.

[AMCP84] P. B. Andrews, D. A. Miller, E. L. Cohen, and F. Pfenning. Automating Higher-Order Logic. In *Automated Theorem Proving: After 25 Years*, pages 169–192. American Mathematical Society, 1984.

[And86] Peter B. Andrews. *An Introduction to Logic and Type Theory: To Truth through Proof.* Academic Press, New York, NY, 1986.

[BB89] Karl Hans Bläsius and Hans-Jürgen Bürckert. *Deduction Systems in Artificial Intelligence.* Ellis Horwood Series in Artificial Intelligence. Ellis Horwood Limited, Chichester, West Sussex, UK, 1989. Distributed in the U.S. by Halsted Press: a division of John Wiley and Sons.

[BC94] R. Bourdeau and B. Cheng. A Formal Semantics of Object Models. Technical Report MSU-CPS-94-6, Department of Computer Science, Michigan State University, East Lansing, Michigan, January 1994.

[BC95a] William R. Bevier and Richard M. Cohen. An Executable Model of the Synergy File System. Technical report, Computational Logic, Inc., May 1995.

[BC95b] R. Bourdeau and B. Cheng. A Formal Semantics for Object Model Diagrams. *IEEE Transactions on Software Engineering*, 21(10):799–821, October 1995.

[BCC$^+$95] Ricky W. Butler, James L. Caldwell, Victor A. Carreno, C. Michael Holloway, Paul S. Miner, and Ben L. Di Vito. NASA Langley's Research and Technology Transfer Program in Formal Methods. In *Tenth Annual Conference on Computer Assurance (COMPASS 95)*, Gaithersburg, MD, June 1995.

[BCM$^+$90] J. R. Burch, E. M. Clarke, K. L McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, PA, June 1990. IEEE Computer Society.

[BDH94] Ricky W. Butler, Ben L. Di Vito, and C. Michael Holloway. Formal Design and Verification of a Reliable Computing Platform for Real-Time Control: Phase 3 Results. NASA Technical Memorandum 109140, NASA Langley Research Center, Hampton, VA, August 1994.

[BE87] Jon Barwise and John Etchemendy. *The Liar: An Essay in Truth and Circularity.* Oxford University Press, New York, NY, 1987.

[BE93] W. Bibel and E. Eder. Methods and Calculi for Deduction. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 1: Logical Foundations, pages 68–182. Oxford, 1993.

[Bee86] Michael J. Beeson. Proving Programs and Programming Proofs. In *International Congress on Logic, Methodology and Philosophy of Science VII*, pages 51–82, Amsterdam, 1986. North-Holland. Proceedings of a meeting held at Salzburg, Austria, in July 1983.

[Bel86] E. T. Bell. *Men of Mathematics.* A Touchstone Book. Simon & Schuster, Inc., New York, NY, 1986. First published in 1937.

[Bev89] William R. Bevier. Kit and the Short Stack. *Journal of Automated Reasoning*, 5(4):519–530, December 1989.

[BG96] Bernard Boigelot and Patrice Godefroid. Symbolic Verification of Communication Protocols with Infinite State Spaces Using QDDs. In Alur and Henzinger [AH96], pages 1–12.

[BH91] J.M. Boyle and T.J. Harmer. Functional Specifications for Mathematical Computations. In B. Möller, editor, *Constructing Programs from Specifications*, pages 205–224. North-Holland, 1991. Proceedings of the IFIP TC2/WG 2.1 Working Conference on Constructing Programs from Specifications, Pacific Grove, CA, USA, 13-16 May 1991.

[BH97] R. Bharadwaj and C. Heitmeyer. Verifying SCR Requirements Specifications using State Exploration. In *First ACM SIGPLAN Workshop on Automatic Analysis of Software*, Paris, France, January 1997. Association for Computing Machinery.

[BHL90] D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors. *VDM '90: VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, Kiel, FRG, April 1990. Springer-Verlag.

[BHMY89] William R. Bevier, Warren A. Hunt, Jr., J Strother Moore, and William D. Young. An Approach to Systems Verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

[BHS91] F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol Specification.* BCS Practitioner Series. Prentice Hall International Ltd., Hemel Hempstead, UK, 1991.

[BJ78] D. Brand and W. H. Joyner, Jr. Verification of Protocols Using Symbolic Execution. *Computer Networks*, 2:351–360, 1978.

[BM79] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, NY, 1979.

[BM88] R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, New York, NY, 1988.

[BMS$^+$96] Ricky W. Butler, Paul S. Miner, Mandayam K. Srivas, Dave A. Greve, and Steven P. Miller. A Bitvectors Library for PVS. NASA Technical Memorandum 110274, NASA Langley Research Center, Hampton, VA, August 1996.

[Boe87] B. Boehm. Industrial Software Metrics Top 10 List. *IEEE Software*, 4(5):84–85, September 1987.

[Boo91] G. Booch. *Object-Oriented Design with Applications.* Benjamin/Cummings Series in Ada and Software Engineering. Benjamin/Cummings Inc., Redwood City, CA, 1991.

[Bowen] http://www.comlab.ox.ac.uk/archive/formal-methods.html. Follow the link “individual notations, methods and tools”.

[Boy89] J.M. Boyle. Abstract Programming and Program Transformations - An Approach to Reusing Programs. In T. J. Biggerstaff and A. J. Perlis, editor, *Software Reusability, Volume I*, pages 361–413. ACM Press, Addison-Wesley Publishing Company, 1989.

[BP83] Paul Benacerraf and Hilary Putnam, editors. *Philosophy of Mathematics: Selected Readings.* Cambridge University Press, Cambridge, England, second edition, 1983.

[BR73] J. W. De Bakker and W. De Roever. A Calculus for Recursive Program Schemes. In M. Nivat, editor, *Automata, Languages, and Programming*, pages 167–196, Amsterdam, 1973. North Holland.

[Bre91] *Algebraic Specification Techniques in Object Oriented Programming Environments*, volume 562 of *Lecture Notes in Computer Science.* Springer-Verlag, 1991.

[Bry86] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[Bry94] Arthur E. Bryson, Jr. *Control of Spacecraft and Aircraft.* Princeton University Press, Princeton, New Jersey, 1994.

[BS93] Jonathan Bowen and Victoria Stavridou. The Industrial Take-up of Formal Methods in Safety-Critical and other Areas: A Perspective. In J. C. P Woodcock and P. G. Larsen, editors, *FME '93: Industrial-Strength Formal Methods*, pages 183–195, Odense, Denmark, April 1993. Volume 670 of *Lecture Notes in Computer Science*, Springer-Verlag.

[Bur69] R. Burstall. Proving Properties of Programs by Structured Induction. *Computing Journal*, 12(1):41–48, 1969.

[Bur84] John P. Burgess. Basic Tense Logic. In Gabbay and Guenthner [GG84], chapter II.2, pages 89–133.

[Bus90] Marilyn Bush. Improving Software Quality: The Use of Formal Inspections at the Jet Propulsion Laboratory. In *12th International Conference on Software Engineering*, pages 196–199, Nice, France, March 1990. IEEE Computer Society.

[BY90] W. R. Bevier and W. D. Young. Machine-Checked Proofs of a Byzantine Agreement Algorithm. Technical Report 55, Computational Logic Incorporated, Austin, TX, June 1990.

[Car58] Rudolf Carnap. *Introduction to Symbolic Logic and Its Applications.* Dover Publications, Inc., New York, NY, 1958. English translation of *Einführung in die symbolische Logik*, 1954.

[CBK90] E. M. Clarke, I. A. Browne, and R. P. Kurshan. A Unified Approach for Showing Language Containment and Equivalence Between Various Types of ω-Automata. In A. Arnold, editor, *CAAP '90, 15th Colloquium on Trees in Algebra and Programming*, pages 103–116, Copenhagen, Denmark, May 1990. Volume 431 of *Lecture Notes in Computer Science*, Springer-Verlag.

[CE81] E. M. Clarke and E. A. Emerson. Characterizing Properties of Parallel Programs as Fixpoints. In *7th International Colloquium on Automata, Languages and Programming.* Volume 85 of *Lecture Notes in Computer Science*, Springer-Verlag, 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CFJ93] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting Symmetry in Temporal Logic Model Checking. In Courcoubetis [Cou93].

[CGK89] E. M. Clarke, O. Grumberg, and R. P. Kurshan. A Synthesis of Two Approaches for Verifying Finite State Concurrent Systems. In A. R. Meyer and M. A. Taitslin, editors, *Logic at Botik '89, Symposium on Logical Foundations of Computer Science*, pages 81–90, Pereslavl-Zalessky, USSR, July 1989. Volume 363 of *Lecture Notes in Computer Science*, Springer-Verlag.

[CGL92] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. In *19th ACM Symposium on Principles of Programming Languages*, pages 343–354, Albuquerque, NM, January 1992. Association for Computing Machinery.

[CH89] Rance Cleaveland and Matthew Hennessy. Testing Equivalence as a Bisimulation Equivalence. In *International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, June 1989. Volume 407 of *Lecture Notes in Computer Science*, Springer-Verlag.

[CHB92] D. Coleman, F. Hayes, and S. Bear. Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design. *IEEE Transactions on Software Engineering*, 18(1):9–18, January 1992.

[CHJ86] B. Cohen, W. T. Harwood, and M. I. Jackson. *The Specification of Complex Systems.* Addison-Wesley, Wokingham, England, 1986.

[CHR92] Zhou Chaochen, C. A. R. Hoare, and A. P. Ravn. A Calculus of Durations. *Information Processing Letters*, 40(5):269–276, 1992.

[CK90] E. M. Clarke and R. P. Kurshan, editors. *Computer-Aided Verification, CAV '90*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science.* American Mathematical Society and Association for Computing Machinery, June 1990.

[CKM$^+$91] Dan Craigen, Sentot Kromodimoeljo, Irwin Meisels, Bill Pase, and Mark Saaltink. EVES: An Overview. In S. Prehn and W. J. Toetenel, editors, *VDM '91: Formal Software Development Methods*, pages 389–405, Noordwijkerhout, The Netherlands, October 1991. Volume 551 of *Lecture Notes in Computer Science*, Springer-Verlag. Volume 1: Conference Contributions.

[CL73] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving.* Computer Science and Applied mathematics. Academic Press, New York, NY, 1973.

[CLM89] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional Model Checking. In *4th Annual IEEE Symposium on Logic in Computer Science*, pages 353–362, Asilomar, Pacific Grove, CA, June 1989. IEEE Computer Society.

[CLS96] David Cyrluk, Patrick Lincoln, and N. Shankar. On Shostak's Decision Procedure for Combinations of Theories. In M. A. McRobbie and J. K. Slaney, editors, *Automated Deduction—CADE-13*, pages 463–477, New Brunswick, NJ, July/August 1996. Volume 1104 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag.

[CLSS96] Rance Cleaveland, Philip Lewis, Scott Smolka, and Oleg Sokolsky. The Concurrency Factory: A Development Environment for Concurrent Systems. In Alur and Henzinger [AH96], pages 398–401.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, 1988.

[CMCHG96] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic Model Checking. In Alur and Henzinger [AH96], pages 419–422.

[Coc84] Nino B. Cocchiarella. Philosophical Perspectives on Quantification in Tense and Modal Logic. In Gabbay and Guenthner [GG84], chapter II.6, pages 309–353.

[COMP95] *COMPASS '95 (Proceedings of the Ninth Annual Conference on Computer Assurance)*, Gaithersburg, MD, June 1995. IEEE Washington Section.

[Cou93] Costas Courcoubetis, editor. *Computer-Aided Verification, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, Elounda, Greece, June/July 1993. Springer-Verlag.

[CPS93] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[CRS96] J. Chang, D. Richardson, and S. Sankar. Structural Specification-Based Testing with ADL. In ISSTA [ISSTA96], pages 62–70.

[CS89] Dan Craigen and Karen Summerskill, editors. *Formal Methods for Trustworthy Computer Systems (FM89)*, Halifax, Nova Scotia, Canada, July 1989. Springer-Verlag Workshops in Computing.

[CS96] Rance Cleaveland and Steve Sims. The NCSU Concurrency Workbench. In Alur and Henzinger [AH96], pages 394–397.

[CVWY92] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yanakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. In *Formal Methods in System Design 1*, pages 275–288. Kluwer, 1992.

[CWB94] B. Cheng, E. Wang, and R. Bourdeau. A Graphical Environment for Formally Developing Object-Oriented Software. In *IEEE International Conference on Tools with AI*, San Diego, CA, November 1994.

[CY91a] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Yourdon Press Computing Series. Prentice Hall, Englewood Cliffs, NJ, 1991. Second Edition.

[CY91b] P. Coad and E. Yourdon. *Object-Oriented Design.* Yourdon Press Computing Series. Prentice Hall, Englewood Cliffs, NJ, 1991.

[Dah90] Ole-Johan Dahl. Object Orientation and Formal Techniques. In D. Bjorner, C.A.R. Hoare, and H. Langmaack, editors, *VDM'90: VDM and Z*, pages 1–11. Number 428 in Lecture Notes in Computer Science, Springer-Verlag, 1990.

[dBdRR89] J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors. *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, REX Workshop, Mook, The Netherlands, May/June 1989. Springer-Verlag.

[dBHdRR91] J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors. *Real Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, REX Workshop, Mook, The Netherlands, June 1991. Springer-Verlag.

[DDHY92] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol Verification as a Hardware Design Aid. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, 1992. Cambridge, MA, October 11-14.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[Dil94] D. L. Dill, editor. *Computer-Aided Verification, CAV '94*, volume 818 of *Lecture Notes in Computer Science*, Stanford, CA, June 1994. Springer-Verlag.

[Dil96] D. Dill. The Murϕ Verification System. In Alur and Henzinger [AH96], pages 390–393.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In van Leeuwen [vL90], chapter 6, pages 243–320.

[DN89] J. Daintith and R. Nelson, editors. *The Penguin Dictionary of Mathematics*. Penguin, London, UK, 1989.

[DR96] Ben L. Di Vito and Larry W. Roberts. Using Formal Methods to Assist in the Requirements Analysis of the Space Shuttle GPS Change Request. NASA Contractor Report 4752, NASA Langley Research Center, Hampton, VA, August 1996.

[DST80] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the Common Subexpressions Problem. *Journal of the ACM*, 27(4):758–771, October 1980.

[EL85] E.A. Emerson and C.L. Lei. Efficient Model Checking in Fragments of the Propositional Mu-Calculus. In *Proceedings of the 10th Symposium on Principles of Programming Languages*, pages 84–96, New Orleans, LA, January 1985. Association for Computing Machinery.

[EL86] E. Allen Emerson and Chin-Laung Lei. Efficient Model Checking in Fragments of the Propositional Mu-Calculus. In *2nd Annual IEEE Symposium on Logic in Computer Science*, pages 267–278. IEEE Computer Society, June 1986.

[ES93] E.A. Emerson and A. Prasad Sistla. Symmetry and Model Checking. In Courcoubetis [Cou93].

[Fag76] M. E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15(3):182–211, March 1976.

[Fag86] M. E. Fagan. Advances in Software Inspection. *IEEE Transactions on Software Engineering*, SE-12(7):744–751, July 1986.

[FBHL84] A. A. Fraenkel, Y. Bar-Hillel, and A. Levy. *Foundations of Set Theory*, volume 67 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, The Netherlands, second printing, second edition, 1984.

[FBWK92] Stuart Faulk, John Brackett, Paul Ward, and James Kirby, Jr. The CoRE Method for Real-Time Requirements. *IEEE Software*, 9(5):22–33, September 1992.

[FC87] S. Faulk and P. Clements. The NRL Software Cost Reduction (SCR) Requirements Specification Methodology. In *Fourth International Workshop on Software Specification and Design*, Monterey, CA, April 1987. IEEE Computer Society.

[FF93] Stephen Fickas and Anthony Finkelstein. Requirements Engineering 1993. In RE [RE93], pages v–vi.

[FKV94] M. D. Fraser, K. Kumar, and V. K. Vaishnavi. Strategies for Incorporating Formal Specifications in Software Development. *Communications of the ACM*, 37(10):74–86, October 1994.

[FMJJ92] J. Fernandez, L. Mounier, C. Jard, and T. Jeron. On-the-fly Verification of Finite Transition Systems. In *Formal Methods in System Design 1*, pages 251–273. Kluwer, 1992.

[FN86] A. Furtado and E. Neuhold. *Formal Techniques for Data Base Design.* Springer-Verlag, 1986.

[Gar84] James W. Garson. Quantification in Modal Logic. In Gabbay and Guenthner [GG84], chapter II.5, pages 249–307.

[GAS89] D.I. Good, R.L. Akers, and L.M. Smith. Report on Gypsy 2.05. Technical Report 1, Computational Logic Inc., Austin, TX, January 1989.

[Gen70] Gerhard Gentzen. *Collected Papers*, edited by M. E. Szabo. Studies in Logic. North Holland, New York, NY, 1970.

[GG83] Dov M. Gabbay and Franz Guenthner, editors. *Handbook of Philosophical Logic–Volume I: Elements of Classical Logic*, volume 164 of *Synthese Library.* D. Reidel Publishing Company, Dordrecht, Holland, 1983.

[GG84] Dov M. Gabbay and Franz Guenthner, editors. *Handbook of Philosophical Logic–Volume II: Extensions of Classical Logic*, volume 165 of *Synthese Library.* D. Reidel Publishing Company, Dordrecht, Holland, 1984.

[Gil60] P. C. Gilmore. A Proof Method for Quantification Theory: Its Justification and Realization. *IBM Journal of Research and Development*, 4:28–35, 1960.

[GKK$^+$88] J. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An Introduction to OBJ3. In *Proceedings of the Conference on Conditional Term Rewriting*, pages 258–263, Orsay, France, 1988. Number 308 in Lecture Notes in Computer Science, Springer-Verlag.

[GL96] P. Godefroid and D. E. Long. Symbolic Protocol Verification with Queue BDDs. In *11th Annual IEEE Symposium on Logic in Computer Science*, pages 198–206, New Brunswick, New Jersey, July 1996. IEEE Computer Society.

[Gla95] James Glanz. Mathematical Logic Flushes Out the Bugs in Chip Designs. *Science*, 267:332–333, January 20, 1995.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, Cambridge, UK, 1993.

[GMT$^+$80] S. L. Gerhart, D. R. Musser, D. H. Thompson, D. A. Baker, R. L. Bates, R. W. Erickson, R. L. London, D. G. Taylor, and D. S. Wile. An Overview of Affirm: A Specification and Verification System. In S. H. Lavington, editor, *Information Processing '80*, pages 343–347, Australia, October 1980. IFIP, North-Holland Publishing Company.

[GMW79] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, 1979.

[GNRR93] Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors. *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, New York, NY, 1993. Springer-Verlag.

[God90] P. Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In Clarke and Kurshan [CK90], pages 321–339.

[God96] *Partial-Order Methods for the Verification of Concurrent Systems, An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science.* Springer-Verlag, 1996.

[Gor] M. J. Gordon. Varieties of Theorem Provers. Unpublished manuscript.

[Gor86] M. Gordon. Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware. In G. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 153–177. Elsevier, 1986. Reprinted in Yoeli [Yoe90, pp. 57–77].

[Gor89] Michael J. C. Gordon. Mechanizing Programming Logics in Higher-Order Logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Theorem Proving*, pages 387–439, New York, NY, 1989. Springer-Verlag.

[GPS96] Patrice Godefroid, Doron Peled, and Mark Staskauskas. Using Partial-Order Methods in the Formal Validation of Industrial Concurrent Programs. *IEEE Transactions on Software Engineering*, 22(7):496–507, July 1996.

[Gro92] RAISE Language Group. *The RAISE Specification Language.* BCS Practitioner Series. Prentice Hall, Hemel Hempstead, UK, 1992.

[GS90] Susanne Graf and Bernhard Steffen. Compositional Minimization of Finite-State Systems. In E. M. Clarke and R. P. Kurshan, editors, *Computer-Aided Verification, CAV '90*, pages 186–196, New Brunswick, NJ, June 1990. Volume 531 of *Lecture Notes in Computer Science*, Springer-Verlag.

[GS93] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math.* Texts and Monographs in Computer Science. Springer-Verlag, New York, NY, 1993.

[H$^+$78] K. L. Heninger et al. Software Requirements for the A-7E Aircraft. NRL Report 3876, Naval Research Laboratory, November 1978.

[H$^+$90] D. Harel et al. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[Haj78] J. Hajek. Automatically Verified Data Transfer Protocols. In *Proceedings of the 4th ICCC*, pages 749–756, Kyoto, Japan, 1978.

[Hal84] Michael Hallett. *Cantorian Set Theory and Limitation of Size.* Number 10 in Oxford Logic Guides. Oxford University Press, Oxford, England, 1984.

[Hal90] A. Hall. Using Z as a Specification Calculus for Object-Oriented Systems. In D. Bjorner, C.A.R. Hoare, and H Langmaack, editors, *VDM'90: VDM and Z*, pages 290–318. Number 428 in Lecture Notes in Computer Science, Springer-Verlag, 1990.

[Har84] David Harel. Dynamic Logic. In Gabbay and Guenthner [GG84], chapter II.10, pages 497–604.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[Hat82] William S. Hatcher. *The Logical Foundations of Mathematics.* Pergamon Press, Oxford, UK, 1982.

[Hay87] Ian Hayes, editor. *Specification Case Studies.* Prentice Hall International Ltd., Hemel Hempstead, UK, 1987.

[Haz83] Allen Hazen. Predicative Logics. In Gabbay and Guenthner [GG83], chapter I.5, pages 331–407.

[HB95a] M. G. Hinchey and J. P. Bowen, editors. *Applications of Formal Methods.* Prentice Hall International Ltd., Hemel Hempstead, UK, 1995. International Series in Computer Science.

[HB95b] M. G. Hinchey and J. P. Bowen. Applications of Formal Methods FAQ. [HB95a], pages 1–15.

[HBGL95] Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. SCR*: A Toolset for Specifying and Analyzing Requirements. In COMP [COMP95], pages 109–122.

[HC91] F. Hayes and D. Coleman. Coherent Models for Object-Oriented Analysis. In *OOPSLA '91 (Object-Oriented Programming Systems, Languages, and Applications 1991) Conference Proceedings*, Phoenix, AZ, October 1991. Communications of the ACM.

[HC95] D. N. Hoover and Zewei Chen. Tablewise, a Decision Table Tool. In COMP [COMP95], pages 97–108.

[HCL95] David Hamilton, Rick Covington, and Alice Lee. Experience Report on Requirements Reliability Engineering Using Formal Methods. In *ISSRE '95: International Conference on Software Reliability Engineering*, Toulouse, France, 1995.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Dataflow Programming Language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[Hen80] K. L. Heninger. Specifying Software Requirements for Complex Systems: New Techniques and Their Application. *IEEE Transactions on Software Engineering*, SE-6(1):2–13, January 1980.

[HFB93] N. Halbwachs, J.-C. Fernandez, and A. Bouajjanni. An executable temporal logic to express safety properties and its connection with the language Lustre. In *Sixth International Symposium on Lucid and Intensional Programming, ISLIP'93, Quebec*, April 1993.

[HGH96] D. N. Hoover, David Guaspari, and Polar Humenn. Applications of Formal Methods to Specification and Safety of Avionics Software. NASA Contractor Report 4723, NASA Langley Research Center, Hampton, VA, April 1996. (Work performed by Odyssey Research Associates).

[HHK96] R. Hardin, Z. Har'El, and R. Kurshan. COSPAN. In Alur and Henzinger [AH96], pages 423–427.

[HJL95] Constance Heitmeyer, Ralph Jeffords, and Bruce Labaw. Tools for Analyzing SCR-Style Requirements Specifications: A Formal Foundation. Technical Report 7499, Naval Research Laboratory, Washington DC, 1995. In press.

[HL94] Constance Heitmeyer and Nancy Lynch. The Generalized Railroad Crossing: A Case Study in Formal Verification of Real-Time Systems. In *Real Time Systems Symposium*, pages 120–131, San Juan, Puerto Rico, December 1994. IEEE Computer Society.

[HLK95] Constance Heitmeyer, Bruce Labaw, and Daniel Kiskis. Consistency Checking of SCR-Style Requirements Specifications. In *International Symposium on Requirements Engineering*, York, England, March 1995. IEEE Computer Society.

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Programming Language Lustre. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992.

[HN96] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering Methodology*, 5(4):293–333, 1996.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, 1985.

[Hol] G. Holzmann. Personnal communication.

[Hol81] G. Holzmann. A Theory for Protocol Validation. In *Proceedings of the First IFIP PSTV Conference on Protocol Specification, Testing, and Verification*, pages 377–391, Teddington, UK, 1981. Also appeared in IEEE Transactions on Computers, C-31(8):730–738, August 1982.

[Hol84] G. Holzmann. Backward Symbolic Execution of Protocols. In *Proceedings of the Fourth IFIP PSTV Conference on Protocol Specification, Testing, and Verification*, pages 19–30, Skytop, PA, 1984.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall Software Series. Prentice-Hall, 1991.

[HP73] P. Hitchcock and D. Park. Induction Rules and Termination Proofs. In M. Nivat, editor, *Automata, Languages, and Programming*, pages 225–251, Amsterdam, 1973. North Holland.

[HP96] G. Holzmann and D. Peled. The State of SPIN. In Alur and Henzinger [AH96], pages 385–389.

[HS96] Klaus Havelund and N. Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In *Formal Methods Europe FME '96*, pages 662–681, Oxford, UK, March 1996. Number 1051 in Lecture Notes in Computer Science, Springer-Verlag.

[Hun87] Warren A. Hunt, Jr. The Mechanical Verification of a Microprocessor Design. Technical Report 6, Computational Logic Incorporated, Austin, TX, 1987.

[ICRE94] *ICRE '94 (Proceedings of the First International Conference on Requirements Engineering)*, Colorado Springs, CO, April 1994. IEEE Computer Society.

[ICRE96] *ICRE '96 (Proceedings of the Second International Conference on Requirements Engineering)*, Colorado Springs, CO, April 1996. IEEE Computer Society.

[ID93] C. Norris Ip and David L. Dill. Better Verification through Symmetry. In *CHDL '93: 11th Conference on Computer Hardware Description Languages and their Applications*, pages 87–100. IFIP, 1993. Ottawa, Canada.

[ID96a] C. Norris Ip and David L. Dill. State Reduction Using Reversible Rules. In *Proceedings of the 33rd Design Automation Conference*, pages 564–567, Las Vegas, NV, June 1996.

[ID96b] C. Norris Ip and David L. Dill. Verifying Systems with Replicated Components in Murphi. In Alur and Henzinger [AH96], pages 147–158.

[IEEE194] *Software Development, IEEE Standard 1498.* IEEE Publications Office, Los Alamitos, CA, March 1994. Interim Standard.

[ISO88] *LOTOS—A Formal Description Technique Based on the Temporal Ordering of Observational Behavior.* International Organization for Standardization—Information Processing Systems—Open Systems Interconnection, Geneva, Switzerland, September 1988. ISO Standard 8807.

[ISSTA96] *ISSTA '96 (Proceedings of the 1996 Symposium on Software Testing and Analysis)*, San Diego, CA, January 1996. Association for Computing Machinery.

[Jac95] M. Jackson. *Software Requirements and Specifications, a lexicon of practice, principles and prejudices.* ACM Press Books. Addison-Wesley, Reading, MA, 1995.

[Jon90] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, second edition, 1990.

[Kay91] Richard Kaye. *Models of Peano Arithmetic.* Number 15 in Oxford Logic Guides. Oxford University Press, Oxford, England, 1991.

[KB70] D. E. Knuth and P. B. Bendix. Simple Word Problems in Universal Algebras. In *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, UK, 1970. Reprinted in *Automation of Reasoning 2* (Springer-Verlag, Berlin, 1983) 342–376.

[KM94] Matt Kaufmann and J Strother Moore. Design Goals for ACL2. Technical Report 101, Computational Logic, Inc., Austin, TX, August 1994.

[KM96] Matt Kaufmann and J Strother Moore. ACL2: An Industrial Strength Version of Nqthm. In *COMPASS '96 (Proceedings of the Eleventh Annual Conference on Computer Assurance)*, pages 23–34, Gaithersburg, MD, June 1996. IEEE Washington Section.

[Knu86] Donald Knuth. *Computers & Typesetting / A: The TEXbook.* Addison-Wesley, Reading, MA, 1986.

[Kow88] R. A. Kowalski. The Early Years of Logic Programming. *Communications of the ACM*, 31(1):38–42, 1988.

[Koz83] Dexter Kozen. Results on the Propositional μ-Calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[Kri63a] Saul Kripke. Semantical Analysis of Modal Logic I, Normal Propositional Calculi. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963. VEB Deutscher Verlag der Wissenschaften, Berlin.

[Kri63b] Saul Kripke. Semantical Considerations on Modal Logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[Kri65] Saul Kripke. Semantical Analysis of Modal Logic II, Non-Normal Modal Propositional Calculi. In J. W. Addison, L. Henkin, and A. Tarski, editors, *The Theory of Models*, pages 206–220. North-Holland, Amsterdam, 1965.

[Kro93] K. Kronöf, editor. *Method Integration: Concepts and Case Studies.* John Wiley & Sons, New York, NY, 1993.

[KSH92] John C. Kelly, Joseph S. Sherif, and Jonathan Hops. An Analysis of Defect Densities Found During Software Inspections. *Journal of Systems Software*, 17:111–117, 1992.

[KTB88] B. Konikowska, A. Tarlecki, and A. Blikle. A Three-Valued Logic for Software Specification and Validation. In R. Bloomfield, L. Marshall, and R. Jones, editors, *VDM'88: VDM — The Way Ahead*, pages 218–242. Number 328 in Lecture Notes in Computer Science, Springer-Verlag, 1988.

[Kur94] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton Series in Computer Science. Princeton University Press, Princeton, NJ, 1994.

[KZ89] D. Kapur and H. Zhang. An Overview of Rewrite Rule Laboratory (RRL). In *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, pages 559–563, Chapel Hill, NC, 1989. Number 355 in Lecture Notes in Computer Science, Springer-Verlag.

[LA94] R. Lutz and Y. Ampo. Experience Report: Using Formal Methods for Requirements Analysis of Critical Spacecraft Software. In *Proceedings of the Nineteenth Annual Software Engineering Workshop, NASA Goddard Space Flight Center*, Greenbelt, MD, December 1994.

[Lam89] Leslie Lamport. A Simple Approach to Specifying Concurrent Systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[Lam95] Leslie Lamport. Types are not Harmless, July 1995. http://www.research.digital.com/SRC/tla/tla.html (under Related Issues).

[Lan96] Christopher Landauer. Discrete Event Systems in Rewriting Logic. In J. Meseguer, editor, *First International Workshop on Rewriting Logic and its Applications*, pages 309–320. Elsevier Science B.V., September 1996. Electronic Notes in Theoretical Computer Science, Volume 4.

[Lev79] Azriel Levy. *Basic Set Theory.* Perspectives in Mathematical Logic. Springer-Verlag, Berlin, Germany, 1979.

[LFB96] Peter Gorm Larsen, John Fitzgerald, and Tom Brookes. Applying Formal Specification in Industry. *IEEE Software*, 13(3):48–56, May 1996.

[LG96] K. Lano and S. Goldsack. Integrating Formal and Object-Oriented Methods: The VDM++ Approach. In *2nd Methods Integration Workshop*, Leeds, UK, 1996. To appear in Springer-Verlag EWTC Series.

[LHHR94] Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

[Lin94] Richard Linger. Cleanroom Process Model. *IEEE Software*, 11(2):50–58, March 1994.

[LO88] E. Lusk and R. Overbeek, editors. *9th International Conference on Automated Deduction (CADE)*, volume 310 of *Lecture Notes in Computer Science*, Argonne, IL, May 1988. Springer-Verlag.

[LPPU94] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. A Framework for Distributed System Designs. In *Proceedings, KBSE '94, The Ninth Knowledge-Based Software Engineering Conference*, pages 48–57, Monterey, California, September 1994. IEEE Computer Society Press.

[LR93a] Patrick Lincoln and John Rushby. Formal Verification of an Algorithm for Interactive Consistency under a Hybrid Fault Model. NASA Contractor Report 4527, NASA Langley Research Center, Hampton, VA, July 1993.

[LR93b] Patrick Lincoln and John Rushby. Formal Verification of an Algorithm for Interactive Consistency under a Hybrid Fault Model. In Courcoubetis [Cou93], pages 292–304.

[LSB92] R. Letz, J. Schumann, and S. Bayerl. SETHEO: A High-Performance Theorem Prover. *Journal of Automated Reasoning*, 8(2):183–212, 1992.

[LSVW96] N. Lynch, R. Segala, F. Vaandrager, and H. B. Weinberg. Hybrid I/O Automata. In Alur et al. [AHS96], pages 496—510.

[Lut93] Robyn R. Lutz. Analyzing Software Requirements Errors in Safety-Critical Embedded Systems. In *IEEE International Symposium on Requirements Engineering*, pages 126–133, San Diego, CA, January 1993.

[Lyn96] N. Lynch. Modeling and Verification of Automated Transit Systems, using Timed Automata, Invariants, and Simulations. In Alur et al. [AHS96], pages 449—463.

[Mac95] Donald MacKenzie. The Automation of Proof: A Historical and Sociological Exploration. *IEEE Annals of the History of Computing*, 17(3):7–29, Fall 1995.

[Man94] Z. Manna. Beyond Model Checking. In Dill [Dil94], pages 220–221.

[MC85] B. Mishra and E. M. Clarke. Hierarchical Verification of Asynchronous Circuits Using Temporal Logic. *Theoretical Computer Science*, 38:269–291, 1985.

[MC94] A. Moreira and R. Clark. Combining Object-Oriented Analysis and Formal Description Techniques. In Tokoro and Pareschi [TP94], pages 344–364.

[McI95] A. McIver. Why Be Formal. *New Scientist Magazine*, pages 34–38, 1995. 26 August.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, MA, 1993.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, 1989.

[Mil93] H. D. Mills. Zero Defect Software: Cleanroom Engineering. In *Advances in Computers*, volume 36, pages 1–41. 1993.

[Min95] Paul S. Miner. Defining the IEEE-854 Floating-Point Standard in PVS. NASA Technical Memorandum 110167, NASA Langley Research Center, Hampton, VA, June 1995.

[ML96] Paul S. Miner and James F. Leathrum, Jr. Verification of IEEE Compliant Subtractive Division Algorithms. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, pages 64–78, Palo Alto, CA, November 1996. Volume 1166 of *Lecture Notes in Computer Science*, Springer-Verlag.

[MLR$^+$96] F. Modugno, N. Leveson, J. Reese, K. Partridge, and S. Sandys. Integrated Safety Analysis of Requirements Specifications. Draft, May 1996.

[MMU83] *MMU Systems Data Book.* NASA MMU-SE-17-73, revision: Basic edition, June 1983. Volume 1 of MMU Operational Data Book.

[Mos85] Ben Moszkowski. A Temporal Logic for Multilevel Reasoning about Hardware. *IEEE Computer*, 18(2):10–19, 1985.

[MPJ94] Paul S. Miner, Shyamsundar Pullela, and Steven D. Johnson. Interaction of Formal Design Systems in the Development of a Fault-Tolerant Clock Synchronization Circuit. Technical Report 405, Computer Science Department, Indiana University, Bloomington, IN, April 1994.

[MS95] Steven P. Miller and Mandayam Srivas. Formal Verification of the AAMP5 Microprocessor: A Case Study in the Industrial Use of Formal Methods. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 2–16, Boca Raton, FL, 1995. IEEE Computer Society.

[Mus80] D. R. Musser. Abstract Data Type Specification in the Affirm System. *IEEE Transactions on Software Engineering*, 6(1):24–32, January 1980.

[MW90] Carroll Morgan and J. C. P. Woodcock, editors. Springer-Verlag Workshops in Computing, January 1990.

[MW95] P. Mukherjee and B. Wichmann. Formal Specification of the STV Algorithm. In Hinchey and Bowen [HB95a], pages 73–96.

[NASA-92] *NASA Software Assurance Standard.* NASA Office of Safety and Mission Assurance, November 1992.

[NASA-93a] *NASA Software Formal Inspections Standard.* NASA Engineering Division Publication, 1993.

[NASA-93b] *NASA Software Formal Inspections Guidebook.* NASA Office of Safety and Mission Assurance, August 1993.

[NASA-95a] *Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Volume I: Planning and Technology Insertion.* NASA Office of Safety and Mission Assurance, Washington, DC, July 1995. NASA-GB-002-95, Release 1.0.

[NASA-95b] *NASA Software Strategic Plan.* National Aeronautics and Space Administration, July 1995.

[NASA-96] *NASA Guidebook for Safety Critical Software Analysis and Development.* NASA Office of Safety and Mission Assurance, April 1996.

[NASA93] *Formal Methods Demonstration Project for Space Applications – Phase I Case Study: Space Shuttle Orbit DAP Jet Select.* Multi-Center NASA Team from Jet Propulsion Laboratory, Johnson Space Center, and Langley Research Center, December 1993. NASA Code Q Final Report (Unnumbered).

[Nay95] P. Pandurang Nayak. *Automated Modeling of Physical Systems.* ACM Distinguished Theses. sv, Berlin, Germany, 1995.

[NO79] G. Nelson and D. C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[Olt95] Walter Olthoff, editor. *9th European Conference on Object-Oriented Programming (ECOOP '95)*, volume 952 of *Lecture Notes in Computer Science*, Åarhus, Denmark, August 1995. Springer-Verlag.

[ORSvH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[OSR93a] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language.* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. A new edition for PVS Version 2 is expected in 1997.

[OSR93b] S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the PVS Specification and Verification System.* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. Three volumes: Language, System, and Prover Reference Manuals; A new edition for PVS Version 2 is expected in late 1997.

[Par70] D. M. R. Park. Fixpoint Induction and Proofs of Program Properties. *Machine Intelligence*, 5, 1970.

[Par76] David Park. Finiteness is Mu-Ineffable. *Theoretical Computer Science*, 3:173–181, 1976.

[Par91] David L. Parnas. *Proposed Standard for Computers in the Safety Systems of Nuclear Power Stations.* Telecommunications Research Institute of Ontario (TRIO), Queen's University, Kingston, Ontario, Canada, March 1991.

[Par95] D. L. Parnas. Using Mathematical Models in the Inspection of Critical Software. In Hinchey and Bowen [HB95a], pages 17–31.

[Pau84] L. C. Paulson. Verifying the Unification Algorithm in LCF. Technical Report 50, University of Cambridge Computer Laboratory, Cambridge, UK, 1984.

[Pau88] Lawrence C. Paulson. Isabelle: The next seven hundred theorem provers. In Lusk and Overbeek [LO88], pages 772–773.

[Pau91] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, Cambridge, UK, 1991.

[Pau92] Lawrence C. Paulson. Designing a Theorem Prover. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science; Volume 2 Background: Computational Structures*, pages 415–475. Oxford Science Publications, Oxford, UK, 1992.

[Pau97] L. C. Paulson. Generic Automatic Proof Tools. In R. Veroff, editor, *Automated Reasoning and Its Applications.* The MIT Press, 1997. To appear.

[Pauls] http://www.cl.cam.ac.uk/users/lcp/hotlist#Systems.

[Pel93] D. Peled. All from One, One for All, on Model-Checking Using Representatives. In Courcoubetis [Cou93], pages 409–423.

[Pel94] D. Peled. Combining Partial Order Reductions with On-the-fly Model-Checking. In Dill [Dil94], pages 377–390.

[Per90] Dominique Perrin. Finite Automata. In van Leeuwen [vL90], chapter 1, pages 1–57.

[PM91] David L. Parnas and Jan Madey. Functional Documentation for Computer Systems Engineering (Version 2). Technical Report TRIO-CRL 237, Telecommunications Research Institute of Ontario (TRIO), Queen's University, Kingston, Ontario, Canada, September 1991.

[Pnu77] A. Pnueli. The Temporal Logic of Programs. In *Proc. 18th Symposium on Foundations of Computer Science*, pages 46–57, Providence, RI, November 1977. ACM.

[PPV60] D. Prawitz, H. Prawitz, and N. Voghera. A Mechanical Proof Procedure and its Realization in an Electronic Computer. *Journal of the Association for Computing Machinery*, 7:102–128, 1960.

[QS82] J. P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in Cesar. In *Proc. 5th International Symposium on Programming*, pages 337–351, Turin, Italy, April 1982. Volume 137 of *Lecture Notes in Computer Science*, Springer-Verlag.

[Que82] J. P. Queille. *Le Système César: Description, Spécification et Analyse des Applications Réparties.* PhD thesis, Computer Science Department, Université de Grenoble, June 1982.

[Rat97] *UML Documentation.* Rational Software Corporation, Santa Clara, CA, 1997. Several documents, including "UML Summary", "UML Notation Guide", and "UML Semantics" are available via the URL http://www.rational.com/uml.

[RB91] J. Rumbaugh and M. Blaha. Tutorial Notes: Object-Oriented Modeling and Design. In *OOPSLA '91 (Object-Oriented Programming Systems, Languages, and Applications 1991) Conference Proceedings*, Phoenix, AZ, October 1991. Communications of the ACM.

[RB96] Larry W. Roberts and Mike Beims. Using Formal Methods to Assist in the Requirements Analysis of the Space Shuttle HAC Change Request (CR90960E). JSC Technical Report, Loral Space Information Systems, Houston, TX, September 1996.

[RBP$^+$91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[RE80] B. Razouk and G. Estrin. Modeling and Verification of Communication Protocols in SARA: the X.21 Interface. *IEEE Transactions on Computers*, C-29(12):1038–1052, 1980.

[RE93] *RE '93 (Proceedings of the IEEE International Symposium on Requirements Engineering)*, San Diego, CA, January 1993. IEEE Computer Society.

[RE95] *RE '95 (Proceedings of the IEEE International Symposium on Requirements Engineering)*, York, England, March 1995. IEEE Computer Society.

[Res69] N. Rescher. *Many-Valued Logic.* McGraw-Hill, New York, NY, 1969.

[RG92] K.S. Rubin and A. Goldberg. Object Behavior Analysis. *Communications of the ACM*, 35(9):48–62, September 1992.

[Rob65] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, January 1965.

[Roc91] *Space Shuttle Orbiter Operational Level C Functional Subsystem Software Requirements: Guidance Navigation and Control—Part C Flight Control Orbit DAP.* Rockwell International, Space Systems Division, OI-21 edition, February 1991.

[RSS95] S. Rajan, N. Shankar, and M.K. Srivas. An Integration of Model-Checking with Automated Proof Checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, pages 84–97, Liege, Belgium, June 1995. Volume 939 of *Lecture Notes in Computer Science*, Springer-Verlag.

[RT52] J. B. Rosser and A. R. Turquette. *Many-Valued Logics*. North-Holland, Amsterdam, Holland, 1952.

[Rus92] John Rushby. Formal Verification of an Oral Messages Algorithm for Interactive Consistency. Technical Report SRI-CSL-92-1, Computer Science Laboratory, SRI International, Menlo Park, CA, July 1992. Also available as NASA Contractor Report 189704, October 1992.

[Rus93a] John Rushby. *Formal Methods and Digital Systems Validation for Airborne Systems*. Federal Aviation Administration Technical Center, Atlantic City, NJ, 1993. Forthcoming chapter for "Digital Systems Validation Handbook," DOT/FAA/CT-88/10.

[Rus93b] John Rushby. Formal Methods and Digital Systems Validation for Airborne Systems. Technical Report SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993. Also available as NASA Contractor Report 4551, December 1993.

[Rus96] John Rushby. Automated Deduction and Formal Methods. In Alur and Henzinger [AH96], pages 169–183.

[RvH93] John Rushby and Friedrich von Henke. Formal Verification of Algorithms for Critical Systems. *IEEE Transactions on Software Engineering*, 19(1):13–23, January 1993.

[RW69] G. Robinson and L. Wos. Paramodulation and Theorem-Proving in First Order Theories with Equality. In *Machine Intelligence*, Edinburgh, Scotland, 1969. Volume 4, Edinburgh University Press.

[SAFER92] Project Requirements Document for the Simplified Aid for EVA Rescue (SAFER) Flight Test Project, December 1992. NASA JSC-24691.

[SAFER94a] *Simplified Aide for EVA Rescue (SAFER)*. NASA JSC-26283, September 1994. Operations Manual.

[SAFER94b] Simplified Aid for EVA Rescue (SAFER) Flight Test Project - Flight Test Article Prime Item Development Specification, July 1994. NASA JSC-25552.

[SB69] D. Scott and J. W. De Bakker. A Theory of Programs. Unpublished Manuscript. IBM. Vienna, 1969.

[SBC92] Susan Stepney, Rosalind Barden, and David Cooper, editors. *Object Orientation in Z.* Workshops in Computing. Springer-Verlag, 1992.

[SD96] Ulrich Stern and David L. Dill. A New Scheme for Memory-Efficient Probabilistic Verification. In *Proceedings of the IFIP TC6/WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols, and Protocol Specification, Testing and Verification*, 1996. To appear.

[SE87] *Software Engineering Standards.* Institute of Electrical and Electronics Engineers, Inc., New York, NY, 1987.

[SG96] D.R. Smith and C.C. Green. Toward Practical Applications of Software Synthesis. In *FMSP'96, The First Workshop on Formal Methods in Software Practice*, pages 31–39, San Diego, CA, January 1996. Association for Computing Machinery.

[SH94] S. Sankar and R. Hayes. Specifying and Testing Software Components Using ADL. Technical Report SMLI TR-94-23, Sun Microsystems Laboratories, Inc.(SMLI), Mountain View, CA, April 1994.

[Sha93] N. Shankar. Abstract Datatypes in PVS. Technical Report SRI-CSL-93-9, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.

[Sha94] N. Shankar. *Metamathematics, Machines, and Gödel's Proof.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, 1994.

[Sha96] Natarajan Shankar. Unifying Verification Paradigms. In Bengt Jonsson and Joachim Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 22–39, Uppsala, Sweden, September 1996. Volume 1135 of *Lecture Notes in Computer Science*, Springer-Verlag.

[Sho67] Joseph R. Shoenfield. *Mathematical Logic.* Addison-Wesley, Reading, MA, 1967.

[Sho78a] J. R. Shoenfield. Axioms of Set Theory. In Jon Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, chapter B1, pages 321–344. North-Holland, Amsterdam, Holland, 1978.

[Sho78b] Robert E. Shostak. An Algorithm for Reasoning about Equality. *Communications of the ACM*, 21(7):583–585, July 1978.

[SM91] S. Shlaer and S. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data.* Yourdon Press Computing Series. Prentice Hall, Englewood Cliffs, NJ, 1991.

[SM95a] Mandayam Srivas and Steven P. Miller. Formal Verification of a Commercial Microprocessor. Technical Report SRI-CSL-95-4, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1995. Also available under the title *Formal Verification of an Avionics Microprocessor* as NASA Contractor Report 4682, July, 1995.

[SM95b] Mandayam K. Srivas and Steven P. Miller. Formal Verification of the AAMP5 Microprocessor. In Hinchey and Bowen [HB95a], chapter 7, pages 125–180.

[Smi90] D.R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, SE-16(9):1024–1043, 1990.

[SOR93] N. Shankar, S. Owre, and J. M. Rushby. *The PVS Proof Checker: A Reference Manual.* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. A new edition for PVS Version 2 is expected in late 1997.

[Spi88] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics.* Cambridge Tracts in Theoretical Computer Science 3. Cambridge University Press, Cambridge, UK, 1988.

[SS86] Leon Sterling and Ehud Shapiro, editors. *The Art of Prolog.* MIT Press Series in Logic Programming. The MIT Press, 1986.

[Sti86] M. E. Stickel. A Prolog Technology Theorem Prover. In J. H. Siekmann, editor, *8th International Conference on Automated Deduction (CADE)*, pages 573–587, Oxford, England, July 1986. Volume 230 of *Lecture Notes in Computer Science*, Springer-Verlag.

[Sys92] ParcPlace Systems. *Object-Oriented Methodology Course Notes.* ParcPlace Systems, Inc., Sunnyvale, CA, 1992.

[Talco] http://www-formal.stanford.edu/clt/ARS/ars-db.html. Follow the link "Existing systems, related fields/pages, archives, ...".

[Tar55] Alfred Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Tar76] Alfred Tarski. *Introduction to Logic and to the Methodology of Deductive Sciences.* Oxford University Press, New York, NY, third edition, 1976. First published 1941.

[Tho84] Richmond H. Thomason. Combinations of Tense and Modality. In Gabbay and Guenthner [GG84], chapter II.3, pages 135–165.

[Tho90] Wolfgang Thomas. Automata on Infinite Objects. In van Leeuwen [vL90], chapter 4, pages 133–187.

[TP94] M. Tokoro and R. Pareschi, editors. *8th European Conference on Object-Oriented Programming (ECOOP '94)*, volume 821 of *Lecture Notes in Computer Science*, Bologna, Italy, July 1994. Springer-Verlag.

[Urq86] A. Urquhart. Many-Valued Logic. In Dov M. Gabbay and Franz Guenthner, editors, *Handbook of Philosophical Logic–Volume III: Alternatives to Classical Logic*, Synthese Library, pages 71–116. D. Reidel Publishing Company, Dordrecht, Holland, 1986.

[Val90] A. Valmari. A Stubborn Attack on State Explosion. In Clarke and Kurshan [CK90], pages 25–42.

[vB88] Johan van Benthem. Time, Logic and Computation. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 1–49, Noordwijkerhout, The Netherlands, May/June 1988. Volume 354 of *Lecture Notes in Computer Science*, Springer-Verlag.

[vBD83] Johan van Bentham and Kees Doets. Higher-Order Logic. In Gabbay and Guenthner [GG83], chapter I.4, pages 275–329.

[vBJ79] L. S. van Benthem Jutting. Checking Landau's 'Grundlagen' in the Automath System. Technical report, Mathematical Centre, Amsterdam, 1979. Mathematical Centre Tracts.

[VDM] http://WWW.ifad.dk/vdm/vdm.html.

[vL90] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics. Elsevier and MIT press, Amsterdam, The Netherlands, and Cambridge, MA, 1990.

[vS90] A. John van Schouwen. The A-7 Requirements Model: Re-Examination for Real-Time Systems and an Application to Monitoring Systems. Technical Report 90-276, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, May 1990.

[vSPM93] A. John van Schouwen, David Lorge Parnas, and Jan Madey. Documentation of Requirements for Computer Systems. In *IEEE International Symposium on Requirements Engineering*, pages 198–207, San Diego, CA, January 1993.

[VW86] M. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *1stAnnual IEEE Symposium on Logic in Computer Science*, Boston, MA, June 1986. IEEE Computer Society.

[Wan60a] H. Wang. Proving Theorems by Pattern Recognition. *Communications of the ACM*, 4(3):229–243, 1960.

[Wan60b] H. Wang. Toward Mechanical Mathematics. *IBM Journal of Research and Development*, 4:2–21, 1960.

[WB93] Trevor Williams and David Baughman. Exploiting Orbital Effects for Short-Range Extravehicular Transfers. In *AAS/AIAA Spaceflight Mechanics Meeting*, Pasadena, CA, 1993. American Astronautical Society.

[WB94] Trevor Williams and David Baughman. Self-Rescue Strategies for EVA Crewmembers Equipped with the SAFER Backpack. In *Proceedings of the Goddard Flight Mechanics/Estimation Theory Symposium*, May 1994. Paper 28.

[Wes78] C. H. West. General Technique for Communications Protocol Validation. *IBM Journal of Research and Development*, 22(3):393–404, 1978.

[Win90] J. M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, 23(9):8–24, September 1990.

[WM85] P. T. Ward and S. J. Mellor. *Structured Development for Real-Time Systems.* Prentice Hall, Englewood Cliffs, NJ, 1985.

[WOLB92] Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning: Introduction and Applications.* McGraw-Hill, New York, NY, second edition, 1992. Includes a copy of the Otter theorem prover for IBM-PCs.

[Wor92] J. B. Wordsworth. *Software Development with Z.* Addison-Wesley, Wokingham, England, 1992.

[wSJGJMW93] John V. Guttag and James J. Horning with S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[WZ78] C. H. West and P. Zafiropulo. Automated Validation of a Communications Protocol: the CCITT X.21 Recommendation. *IBM Journal of Research Development*, 22(1):60–71, 1978.

[Yoe90] Michael Yoeli, editor. *Formal Verification of Hardware Design.* IEEE Computer Society, Los Alamitos, CA, 1990.

[You95] William D. Young. Modelling and Verification of a Simple Real-Time Railroad Gate Controller. In Hinchey and Bowen [HB95a], chapter 8, pages 181–201.

[Zav95] Pamela Zave. Classification of Research Efforts in Requirements Engineering. In RE [RE95], pages 214–216.

Appendix A

Glossary of Key Terms

This appendix contains an alphabetized list of the acronyms and key terms used in the body of the Guidebook.

A.1 Acronyms

AAH: Automatic Attitude Hold
CEA: Control Electronics Assembly of the MMU
CTL: Computational Tree Logic
DCU: Display and Control Unit
DRA: Data Recorder Assembly
EMU: Extravehicular Mobility Unit
EVA: Extravehicular Activity
FOL: First-Order (Predicate) Logic
FSSR: Functional Subsystem Software Requirements
GPS: Global Positioning System
HCM: Hand Controller Module
HCU: Hand Controller Unit
HHMU: Hand Held Maneuvering Unit
IRU: Inertial Reference Unit
LCD: Liquid Crystal Display
LED: Light Emitting Diode
LTL: Linear Temporal Logic
MIR: Mode Identification and Reconstruction
MMU: Manned Maneuvering Unit
OMT: Object Modeling Technique
PLSS: Primary Life Support Subsystem
PSA: Power Supply Assembly
PVS: Prototype Verification System
RHC: Rotational Hand Control of the MMU

ROT: Rotational
SAFER: Simplified Aid for EVA
SCR: Software Cost Reduction (Methodology)
TCC: Type Correctness Conditions
THC: Translational Hand Control of the MMU
TRAN: Translational
VDA: Valve Drive Assemblies

A.2 Terms[1]

assurance: Those activities that demonstrate the conformance of a product or process to a specified criterion such as a functional requirement. *Quality assurance* refers to those activities that focus particularly on conformance to standards or procedures [NASA-92].

axiom: A statement or well-formed formula that is stipulated or assumed rather than proved to be true through the application of rules of inference. The axioms and the rules of inference together provide a basis for proving all other theorems. Axioms are typically identified as *logical* or *nonlogical*. The latter deal with specific domain information, while the former characterize logical properties. A given formal system may have several (different) *axiomatizations*.

formal logic: The study of deductive argument that focuses on the form, rather than the content of the argument. The central concept of formal logic is that of a *valid* argument: if the premises are true, the conclusion must also be true.

formal methods: A varied set of techniques from formal logic and discrete mathematics used in the design, specification, and verification of computer systems and software.

function: A rule f that assigns to every element x of a set X, a unique element y of a set Y, written $y = f(x)$. X is called the *domain* and Y the *range* (or *codomain*). For example, the area of a circle, y, is a function of the radius, x, written $y = f(x) = \pi r^2$. A function with domain X and range Y is also called a *mapping* or *map* from X to Y, written $f : X \rightarrow Y$. A function that maps every element of its domain to an element in its range is said to be *total*. A function that maps some elements of its domain to elements of its range, leaving others undefined, is said to be *partial*.

functional: A function that takes a set of functions as domain and a set of functions as range. For example, the differential operator d/dx is a functional of differentiable functions $f(x)$.

---

[1] Material from [DN89] has been used in some of the following definitions.

model: (1) In logic, an interpretation, I, of a set of well-formed formulas of a formal language such that each member of the set is true in I. (2) A system of definitions, assumptions, and equations set up to represent and discuss physical phenomena and systems.

model theory: The study of the interpretations (models) of formal systems, especially the notions of logical consequence, validity, completeness, and soundness.

mu-Calculus: The μ-calculus is a quantified Boolean logic with least and greatest fixed-point operators.

parsing: A form of analysis that detects syntactic inconsistencies and anomalies, including misspelled keywords, missing delimiters, and unbalanced brackets or parentheses.

power set: The power set of a set A is the set of all sets included in A. If a set has n elements, its power set will have 2^n elements. For example, if a set $S = \{a, b\}$, then the power set of S, $\mathcal{P}(S)$, is the set $\{\emptyset, \{a\}, \{b\}, \{a, b\}\}$.

proof: A chain of reasoning using rules of inference and a set of axioms that leads to a conclusion.

proof theory: The study of proofs and provability in formal languages, including notions of deducibility, independence, simple completeness, and, particularly, consistency.

quantifier: A logical operator that binds a variable in a logical formula and is used to indicate the *quantity* of a proposition, for example, the *univeral* quantifier $\forall$ (read "for all"), and the *existential* quantifier $\exists$ (read "there exists").

requirements: The set of conditions or essential, necessary, or desired capabilities that must be met by a system or system component to satisfy a contract, standard, or other formally implied document or description.

rule of inference: A rule in logic that defines the reasoning that determines when a conclusion may be drawn from a set of premises. In a formal system, the rules of inference should guarantee that *if* the premises are true, then the conclusion is also true.

specification (formal): A characterization of a planned or existing system expressed in a formal language.

testing: Process of exercising or evaluating software by manual or automated means to demonstrate that it satisfies specified requirements or to identify differences between expected and actual results [NASA-92].

trace: A function from *time* to a given type of value, where time represents, for example, a frame, cycle, or iteration count.

typechecking: A form of analysis that detects semantic inconsistencies and anomalies, including undeclared names and ambiguous types.

validation: The process by which a delivered system is demonstrated to satisfy its requirements by testing it in execution. Informally, demonstrating that the requirements are right.

verification: The process of determining whether each level of a specification, and the final system itself, fully and exclusively implements the requirements of its superior specification. Informally, demonstrating that a system is built according to its requirements.

Appendix B

Further Reading

This appendix contains suggestions for further reading, arranged by topic.

B.1 Technical Background: Mathematical Logic

- Peter B. Andrews. *An Introduction to Logic and Type Theory: To Truth through Proof*. Academic Press, New York, NY, 1986.
- Jon Barwise. "An Introduction to First-order Logic." In Jon Barwise, editor, *Handbook of Mathematical Logic*, Volume 90 of *Studies in Logic and the Foundations of Mathematics*, Chapter A1, pages 5–46. North-Holland, Amsterdam, Holland, 1978.
- H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.
- David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math.* Texts and Monographs in Computer Science. Springer-Verlag, New York, NY, 1993.
- Dov M. Gabbay and Franz Guenthner, editors. *Handbook of Philosophical Logic–Volume I: Elements of Classical Logic*, volume 164 of *Synthese Library.* D. Reidel Publishing Company, Dordrecht, Holland, 1983.
- Dov M. Gabbay and Franz Guenthner, editors. *Handbook of Philosophical Logic–Volume II: Extensions of Classical Logic*, volume 165 of *Synthese Library.* D. Reidel Publishing Company, Dordrecht, Holland, 1984.
- Elliott Mendelson. *Introduction to Mathematical Logic.* D. Van Nostrand Company, The University Series in Undergraduate Mathematics, 1964.
- Mark Ryan and Martin Sadler. "Valuation Systems and Consequence Relations." In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science; Volume 1 Background: Mathematical Structures*, pages 1–78. Oxford Science Publications, Oxford, UK, 1992.

- Joseph R. Shoenfield. *Mathematical Logic.* Addison-Wesley, Reading, MA, 1967.
- Johan van Bentham and Kees Doets. Higher-order Logic. In Dov M. Gabbay and Franz Guenthner, editors. *Handbook of Philosophical Logic–Volume I: Elements of Classical Logic*, Chapter I.4. Synthese Library, D. Reidel, 1983, pages 275–329.

B.2 Specification

- J. P. Bowen. *Formal Specification and Documentation Using Z.* International Thomson Computer Press, 1996.
- Dines Bjørner and Cliff B. Jones. *Formal Specification and Software Development.* Prentice Hall International Series in Computer Science, 1986.
- John V. Guttag and James J. Horning with S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.
- Ian Hayes, editor. *Specification Case Studies.* Prentice Hall International Ltd., 1987.
- Michael Hinchey and S. A. Jarvis. *Concurrent Systems: Formal Development in CSP.* McGraw-Hill International Series in Software Engineering, 1995.
- Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice Hall International Series in Computer Science, second edition, 1990.
- Kevin Lano. *The B Language and Method: A Guide to Practical Formal Development.* Springer-Verlag FACIT Series, May 1996.
- J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics.* Cambridge Tracts in Theoretical Computer Science 3. Cambridge University Press, 1988.

B.3 Model Checking

- Edmund Clarke and Robert Kurshan. “Computer-Aided Verification.” *IEEE Spectrum*, Volume 33, No. 6, June 1996, pages 61-67.
- G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall Software Series, 1991.
- Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton Series in Computer Science. Princeton University Press, 1994.
- Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

B.4 Theorem Proving

- R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, 1979.
- R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, 1988.
- Alan Bundy. *The Computer Modelling of Mathematical Reasoning.* Academic Press, 1983.
- M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, 1993.
- Lawrence Paulson. "Designing a Theorem Prover." In S. Abramsky and Dov M. Gabbay and T. S. E. Maibaum, *Handbook of Logic in Computer Science; Volume 2 Background: Computational Structures.* Oxford Science Publications, Oxford, UK, 1992, pages 415–475.
- Larry Wos and Ross Overbeek and Ewing Lusk and Jim Boyle. *Automated Reasoning: Introduction and Applications*, McGraw-Hill, 1992.

B.5 Models of Computation

- C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International Series in Computer Science, 1985.
- K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.
- E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.
- D. Gries. *The Science of Programming.* Springer-Verlag, 1981.
- R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
- Zohar Manna and Richard Waldinger. *The Deductive Foundations of Computer Programming.* Addison-Wesley, 1993.

B.6 Applications and Overviews

- Edmund Clarke and Jeannette Wing. *Formal Methods: State of the Art and Future Directions.* Report of the ACM Workshop on Strategic Directions in Computing Research, Formal Methods Subgroup. Available as Carnegie Mellon University Technical Report CMU-CS-96-178, August 1996.

- Dan Craigen, Susan Gerhart, and Ted Ralston. *An International Survey of Industrial Applications of Formal Methods; Volume 1: Purpose, Approach, Analysis and Conclusions; Volume 2: Case Studies.* National Institute of Standards and Technology, NIST GCR 93/626, 1993.
- C. Neville Dean and Michael Hinchey, eds. *Teaching and Learning Formal Methods.* Academic Press, International Series in Formal Methods, 1996.
- M. G. Hinchey and J. P. Bowen, editors. *Applications of Formal Methods.* Prentice Hall International Series in Computer Science, 1995.
- *IEEE Computer*, Special Issue on Formal Methods. Volume 23, Number 9, September, 1990.
- *IEEE Software*, Special Issue on Formal Methods. Volume 7, Number 5, September, 1990.
- *IEEE Transactions on Software Engineering*, Special Issue on Formal Methods in Software Engineering. Volume 16, Number 9, September, 1990.
- H. Saiedian, ed. "An Invitation to Formal Methods." *IEEE Computer*, Volume 29, Number 4, April 1996, pages 16-30.

B.7 Tutorials

- Ricky W. Butler. *An Introduction to Requirements Capture Using PVS: Specification of a Simple Autopilot.* NASA Langley Research Center, NASA Technical Memorandum 110255, 1996.
- J. Crow and S. Owre and J. Rushby and N. Shankar and M. Srivas. "A Tutorial Introduction to PVS." Presented at IEEE Computer Society Workshop on Industrial-Strength Formal Specification Techniques (WIFT'95), Boca Raton, Florida, 1995.
- Stuart R. Faulk. *Software Requirements: A Tutorial.* Naval Research Laboratory. NRL Memorandum Report No. 5546-95-7775, November, 1995.
- Chris George. "The RAISE Specification Language: A Tutorial," In *VDM '91: Formal Software Development Methods*, S. Prehn and W. J. Toetenel, editors. Springer-Verlag Lecture Notes in Computer Science, Volume 552, October 1991, pages 238–319.
- John Rushby and David W. J. Stringer-Calvert. *A Less Elementary Tutorial for the PVS Specification and Verification System*, SRI, International Technical Report No. SRI-CSL-95-10, July 1996.

- V. Stavridou and A. Boothroyd and P. Bradley and B. Dutertre and L. Shackleton and R. Smith. “Formal Methods and Safety Critical Systems in Practice.” In *High Integrity Systems*, Volume 1, No. 5, 1996, pages 423–445.
- Debora Weber-Wulff. “Proof Movie—A Proof with the Boyer-Moore Prover.” In *Formal Aspects of Computing*, Volume 5, No. 2, 1993, pages 121–151.

Appendix C

Extended Example: Simplified Aid for EVA Rescue (SAFER)

The example presented in this appendix is based on NASA's Simplified Aid for EVA Rescue (SAFER). SAFER is a new system for free-floating astronaut propulsion that is intended for use on Space Shuttle missions, as well as during Space Station construction and operation. Although the specification attempts to capture as much as possible of the actual SAFER design, certain pragmatically motivated deviations have been unavoidable. Nevertheless, the SAFER example contains elements typical of many space vehicles and the computerized systems needed to control them.

C.1 Overview of SAFER

SAFER is a small, self-contained, backpack propulsion system enabling free-flying mobility for a crewmember engaged in extravehicular activity (EVA) that has evolved as a streamlined version of NASA's earlier Manned Maneuvering Unit (MMU) [MMU83]. SAFER is a single-string system designed for contingency use only. SAFER offers sufficient propellant and control authority to stabilize and return a tumbling or separated crewmember, but lacks the propellant capacity and systems redundancy provided with the MMU. Nevertheless, SAFER and the MMU share an overall system concept, as well as general subsystem features. The description that follows draws heavily on the SAFER Operations Manual [SAFER94a] and on the SAFER Flight Text Project development specification [SAFER94b], excerpts of which have been included here as appropriate.

C.1.1 History, Mission Context, and System Description

SAFER is designed as a self-rescue device for a separated EVA crewmember in situations where the Shuttle Orbiter is unavailable to effect a rescue. Typical situations include whenever the Orbiter is docked to a large payload or structure, such as the Russian Mir Space Station or the International Space Station Alpha. A SAFER device would be worn by every crewmember during these types of EVAs. As noted in [WB94], a crewmember engaged in EVA, who becomes separated from an Orbiter or a space station, has three basic options: grappling the Orbiter or station immediately using a "shepherd's crook" device, rescue by a second crewmember flying an MMU (Manned Maneuvering Unit)[1] or self-rescue using a propulsive system. The first option is not realistic in all situations; it assumes a near-optimal response by a tumbling astronaut. The second option is also unrealistic, in this case because it assumes constant availability of both the MMU and the second crewmember during all EVA, since reaction time is critical to successful rendezvous with a drifting crewmember. The third option, a propulsive self-rescue system, is the most viable and therefore the one ultimately selected.

As described in [WB93], the simplest self-rescue system is the Hand-Held Maneuvering Unit (HHMU) or "gas gun" flown on Gemini and Skylab, and the "Crew Propulsive Device," a redesign of the Gemini HHMU flown on the STS-49. Tests on these units indicated that the HHMUs were adequate for short translations, but required the crewmember to visually determine and effectively nullify tumble rates about all three axes – a challenging proposition even with good visual cues. As a result, recommendations based on the STS-49 tests included an automatic detumble capability for all self-rescue devices.

While the HHMU lacked automatic detumble and other capabilities necessary for a general-purpose self-rescue system, the MMU was too well-endowed. The MMU performed the first self-propelled untethered EVA during the STS-41B mission in 1984, participated in the Solar Maximum Mission spacecraft repair on a subsequent 1984 shuttle flight, and was used to capture the Palapa B-2 and the Westar-VI communications satellites on yet another shuttle flight that same year [WB94, p. 4]. However, the MMU's versatility, redundancy, and physical bulk made it unsuited as a general-purpose self-rescue device. Nevertheless, so many MMU features have been incorporated into SAFER (ranging from the hand controller grip design to the gaseous-nitrogen (GN_2) recharge-port quick-disconnect and the GN_2pressure regulator/relief valve), that SAFER has been described as a "mimimized derivative" of the MMU [WB94, p. 2].

SAFER fits around the Extravehicular Mobility Unit (EMU) primary life support subsystem (PLSS) backpack without limiting suit mobility (Figure C.1). SAFER uses 24 GN_2thrusters to achieve six degree-of-freedom maneuvering control. A single hand controller attached to the EMU display and control module is used to control SAFER operations. Propulsion is available either on demand, that is, in response to hand controller inputs, or through an automatic attitude hold (AAH) capability. Hand controller inputs can command either translations or rotations, while attitude hold is designed to bring and keep rotation rates close to zero. SAFER's propulsion system can be recharged during an EVA in the Orbiter payload bay. The SAFER unit weighs approximately 85 pounds and folds for launch, landing, and on-orbit stowage inside the Orbiter airlock.

---

[1] Or, similarly, by a robotic-controlled MMU. However, such a system has apparently not yet been developed and is not likely to be available in the near-term.

C.1.2 Principal Hardware Components

The SAFER flight unit consists of three assemblies: the backpack propulsion module, the hand controller module (HCM), and a replaceable battery pack. SAFER also requires several items of flight support equipment during a Shuttle mission. For the purpose of this discussion, only the propulsion and hand controller modules need be included.

C.1.2.1 Backpack Propulsion Module

The propulsion module is the primary assembly of the SAFER system, attaching directly to the EMU PLSS backpack. Figure C.2 shows the structures and mechanisms contained within the propulsion module. Four subassemblies are identified: main frame structure, left and right tower assemblies, and the avionics box. A lightweight, aluminum-alloy frame holds SAFER components, while external surfaces are formed by an outer aluminum skin. With the exception of the upper thrusters mounted to the tower assemblies, all propulsion subsystem components are housed in the main frame.

The tower assemblies have hinge joints that allow them to be folded for stowage. Towers are unfolded and attached to PLSS interfaces in preparation for an EVA. Latches on the towers hold SAFER firmly to the PLSS. Hinge joints accommodate GN_2 tubing, electrical power, and signal routing to the upper thrusters.

Housed in the avionics box are the control electronics assembly, inertial reference unit, data recorder assembly, and power supply assembly. The avionics box is attached to the bottom of the main frame, as depicted in Figure C.2. Data and power connectors provide an interface to the main frame. Connectors are also provided for the HCM umbilical and ground servicing equipment.

Within the main frame, high-pressure GN_2 is stored in four cylindrical stainless-steel tanks. Pressure and temperature sensors are placed directly adjacent to the tanks and these parameters are displayed to the SAFER crewmember on the HCM. Other components attached to the main GN_2 line are a manual isolation valve, a quick-disconnect recharge port, an integrated pressure regulator and relief valve, and downstream pressure and temperature sensors.

After passing through the regulator/relief valve, GN_2 is routed to four thruster manifolds, each containing six electric-solenoid thruster valves. A total of 24 thrusters is provided, with four thrusters pointing in each of the ±X, ±Y, and ±Z directions. Thruster valves open when commanded by the avionics subsystem. When a valve opens, GN_2 is released and expanded through the thruster's conical nozzle to provide a propulsive force. The avionics subsystem can command as many as four thrusters at a time to provide motion with six degrees of freedom (±X, ±Y, ±Z, ±roll, ±pitch, and ±yaw). Figure C.3 illustrates the thruster layout, designations, and directions of force.

C.1.2.2 Hand Controller Module (HCM)

A SAFER crewmember controls the flight unit and monitors its status by means of the hand controller module (HCM). Two distinct units are found in the HCM: a display and control unit, and a hand controller unit. Both units are mounted together, as shown in Figure C.4, with an internal connector joining the two units electrically.

Various displays and switches are located on the display and control unit and positioned so that they can be viewed from any head position within the EMU helmet. These displays and switches include

1. **Liquid crystal display.** A 16-character, backlit LCD displays prompts, status information, and fault messages to the crewmember.

2. **Thruster cue light.** A red LED lights whenever a thruster-on condition is detected by the control software. This light is labeled "THR."

3. **Automatic attitude hold light.** A green LED labeled "AAH" lights whenever attitude hold is enabled for one or more rotational axes.

4. **Power/test switch.** A three-position toggle switch labeled "PWR" is used to power on the flight unit and initiate self-test functions. The three positions are "OFF," "ON," and "TST."

5. **Display proceed switch.** A three-position, momentary-contact toggle switch is used to control message displays on the LCD. This switch, which is labeled "DISP" on the HCM, is normally in the center null position. When pushed up/down, the switch causes the LCD to display the previous/next parameter or message.

6. **Control mode switch.** A two-position toggle switch is used to configure the hand controller for either rotational or translational commands. This switch is labeled "MODE," with its two positions labeled "ROT" and "TRAN."

The hand controller grip is compatible with an EMU glove. It is mounted on the right side of the HCM with an integral push-button switch for initiating and terminating AAH mode. A four-axis mechanism having three rotary axes and one transverse axis is the heart of the hand controller. A command is generated by moving the grip from the center null position to mechanical hardstops on the hand controller axes. Commands are terminated by deliberately returning the grip to its center position or by releasing the grip so that it automatically springs back to the center.

As shown in Figure C.5, with the control mode switch in the TRAN position, ±X, ±Y, ±Z, and ±pitch commands are available. ±X commands are generated by rotating the grip forward or backward, ±Y commands by pulling or pushing the grip right or left, and ±Z commands by rotating the grip down or up. ±pitch commands are generated by twisting the grip up or down about the hand controller transverse axis.

As shown in Figure C.6, with the control mode switch in the ROT position, ±roll, ±pitch, ±yaw, and ±X commands are available. ±roll commands are generated by rotating the grip down or up (same motion as the ±Z commands in TRAN mode). ±yaw commands are generated by pulling or pushing the grip right or left (same motion as the ±Y commands in TRAN mode). The ±pitch and ±X commands are generated as in TRAN mode, thus making them available in both TRAN and ROT modes.

An electrical umbilical connects the HCM to the propulsion module, attaching to a connector on the avionics box. This umbilical is connected prior to launch and is not intended to be disconnected in flight.

C.1.2.3 Battery Pack

The battery pack, which provides power for all SAFER electrical components, connects to the bottom of the propulsion module, as shown in Figure C.2. Two separate battery circuits are found in the battery pack, both containing multiple stacks of 9-volt alkaline batteries. One battery circuit powers the thruster valves, offering 30–57 volts to the power supply assembly, which produces a 28-volt output for opening valves in pulses of 4.5 milliseconds duration. Energy capacity is sufficient to open the thrusters 1200 times and thereby drain the GN_2 tanks four times. The other battery circuit powers the avionics subsystem (i.e., the remaining electrical components), producing 16–38 volts for the power supply for a cumulative power-on time of 45 minutes. A temperature sensor in the battery pack is used for monitoring purposes. Flight procedures allow for battery pack changing during an EVA.

C.1.2.4 Flight Support Equipment

Besides the SAFER flight unit, several types of flight support equipment are needed during SAFER operations. These items include a special plug to attach the hand controller module to the EMU display and control module, a recharge hose for GN_2 tank recharging during an EVA, the Orbiter's GN_2 system to provide GN_2 via the recharge hose, a SAFER recharge station having handrails and foot restraints to facilitate the recharging procedure, an airlock stowage bag for storing SAFER when not in use, and a battery transfer bag for storing extra battery packs during an EVA. None of these support items will be considered any further in this appendix.

C.1.3 Avionics

SAFER's avionics subsystem resides mostly in the backpack module beneath the propulsion components. Included are the following assemblies:

1. **Control Electronics Assembly (CEA).** Found in the avionics box, the CEA contains a microprocessor that takes inputs from sensors and hand controller switches, and actuates the appropriate thruster valves. The CEA has a serial bus interface for the HCM umbilical as well as an interface for ground support equipment.

2. **Inertial Reference Unit (IRU).** Central to the attitude hold function, the IRU senses angular rates and linear accelerations. Three quartz rate sensors, rate

noise filters, and associated rate measurement electronics provide angular rate sensing up to ±30 degrees per second. A separate sensor exists for each angular axis (roll, pitch, yaw). In addition, a temperature sensor is paired with each of the three rate sensors, enabling the avionics software to reduce rate sensor error caused by temperature changes. An accelerometer senses linear acceleration up to ±1 g along each linear axis (X, Y, Z). These accelerations are recorded by the data recorder assembly for post-flight analysis.

3. **Data Recorder Assembly (DRA).** SAFER flight performance data is collected by the DRA. Saved parameters include data from rate sensors, accelerometers, pressure and temperature transducers, and battery voltage sensors. The DRA also records hand controller and AAH commands and thruster firings. Data may be recorded at one of three rates: 1 Hz, 50 Hz, or 250 Hz. A suitable rate is chosen automatically based on which control mode is in use.

4. **Valve Drive Assemblies (VDAs).** Four valve drive assemblies are used to actuate the GN_2 thrusters. A VDA is located with each cluster of six thrusters (in each tower and on the left and right sides of the propulsion module main frame). VDAs accept firing commands from the CEA and apply voltages to the selected valves. VDAs also sense current flow through the thruster solenoids, providing a discrete signal to the CEA acknowledging thruster firing.

5. **Power Supply Assembly (PSA).** Regulated electrical power for all SAFER electrical components is produced by the PSA. Two battery circuits provide input power, and the PSA serves as a single-point ground for all digital and analog signal returns.

6. **Instrumentation Electronics.** A variety of sensors is included in the SAFER instrumentation electronics. A subset of the sensed parameters is available for display by the crewmember. Table C.1 lists all the SAFER sensors.

C.1.4 System Software

The avionics software is responsible for controlling the SAFER unit in response to crewmember commands. Two principal subsystems comprise the system software: the maneuvering control subsystem and the fault detection subsystem. Maneuvering control includes both commanded accelerations and automatic attitude hold actions. Fault detection supports in-flight operation, pre-EVA checkout, and ground checkout.

C.1.4.1 Software Interfaces

Digital interfaces to SAFER components enable the CEA's microprocessor to achieve control. Four classes of inputs are monitored and accepted by the avionics software:

Parameter measured	Sensor type	Displayed?
GN_2 tank pressure	Pressure	Y
GN_2 tank temperature	Temperature	Y
GN_2 regulator pressure	Pressure	Y
GN_2 regulator temperature	Temperature	Y
Roll rate	Angular rate	Y
Pitch rate	Angular rate	Y
Yaw rate	Angular rate	Y
Electronics battery volts	Voltage	Y
Valve drive battery volts	Voltage	Y
Battery temperature	Temperature	Y
X acceleration	Linear acceleration	N
Y acceleration	Linear acceleration	N
Z acceleration	Linear acceleration	N
Roll rate sensor temperature	Temperature	N
Pitch rate sensor temperature	Temperature	N
Yaw rate sensor temperature	Temperature	N

Table C.1: SAFER sensor complement.

1. **Hand controller switches.** Indications of switch operation cover both toggle switches and those embedded within the hand grip mechanism.

2. **Avionics transducers.** Sensor inputs are converted from analog to digital form before software sampling.

3. **Thruster-on discrete.** This input is a binary indication of at least one thruster valve being open.

4. **Serial line.** Ground checkout operations send data through this input.

Similarly, four classes of outputs are generated by the avionics software:

1. **Hand controller displays.** Both LEDs and a 16-character LCD display are included to present status to the crewmember.

2. **Thruster system.** Digital outputs are delivered to the valve drive assemblies to actuate individual thruster valves.

3. **Data recorder system.** Selected data items are recorded for post-flight analysis on the ground.

4. **Serial line.** Ground checkout operations receive data through this output.

C.1.4.2 Maneuvering Control Subsystem

Figure C.7 breaks down the SAFER software architecture in terms of its primary modules. Those modules comprising the maneuvering control subsystem collectively realize SAFER's six degree-of-freedom propulsion capability. Both rotational and translational accelerations will be commanded by the software. Rotations resulting from the AAH function are invoked automatically by the software in response to rotation rates sensed by the inertial reference unit. Special cases result from the interaction of the AAH function and explicitly commanded accelerations.

Translation commands from the crewmember are prioritized so that only one translational axis receives acceleration, with the priority order being X, Y, and then Z. Whenever possible, acceleration is provided as long as a hand controller or AAH command is present. If both translation and rotation commands are present simultaneously, rotation takes priority and translations will be suppressed. Conflicting input commands result in no output to the thrusters.

The SAFER crewmember can initiate AAH at any time by depressing or "clicking" the pushbutton on the hand controller grip. Whenever AAH is active in any axis the green LED on the HCM illuminates. When the button is double clicked (two clicks within a 0.5 second interval), AAH is disabled for all three rotational axes. If AAH is active, and the crewmember issues a rotational acceleration command about any axis, AAH is immediately disabled on that axis. When this occurs, the remaining axes remain in AAH. On the other hand, if AAH is initiated simultaneously with a rotational command from the hand controller, the rotational command will be ignored and AAH will become active in that axis. This feature is necessary so that a failed-on HCM rotational command cannot permanently disable AAH on the affected axis.

AAH implements an automatic rotational deceleration sufficient to reduce axis rates to near-zero levels. Continuous thruster firings are commanded to reduce rotation about an axis whenever its rate is sensed to be above 0.2 degree per second. Once the rates have fallen below 0.3 degree per second, thrusters are fired only as needed to maintain attitude within approximately ±5 degrees. Thrusters are not fired when attitude is within a ±2 degree deadband.

Rate sensors, rate noise filters, and associated rate measurement electronics exhibit significant offset errors, which are largely a function of rate sensor temperature. Offset reduction is used to minimize the negative effects of rate offset errors. Temperature measurements are periodically sampled and net offset errors estimated. Such estimates are subtracted from the noise filter rate measurements to minimize rate offset errors. Net offset errors are independent for each axis, reaching an average of 0.2 degree per second and resulting in an average drift of the same magnitude.

Acceleration commands from the hand controller and from the AAH function are combined to create a single acceleration command. Thruster select logic is provided to choose suitable thruster firings to achieve the commanded acceleration. Thruster selection results in on-off commands for each thruster, with a maximum of four thrusters turned on simultaneously. Thruster arrangement and designations are shown in Figure C.3, while Tables C.2 and C.3 specify the selection logic. These tables are specified in terms of three possible command values for each axis: negative thrust, positive thrust, or no thrust at all.

C.1.4.3 Fault Detection Subsystem

The fault detection subsystem performs four testing functions: a self test, an activation test, a monitoring function, and a ground checkout function. The fault detection subsystem also manages the display interface, performing the computation of parameters and construction of messages for the HCM LCD.

The self test provides an overall functional test of the SAFER flight unit without using any propellant or external equipment. To carry out the test, the crewmember is led through a checklist of prompts displayed on the HCM LCD. If a particular test is unsuccessful, a failure message is displayed. The following tests are performed during self test:

1. Thruster test
2. Hand controller controls and display test
3. Rate sensor function test

The activation test checks the functionality of the SAFER flight unit in an operational mode, being invoked to check the function of the pressure regulator. A minimal amount of propellant is used and no external equipment is required. The test consists of commanding 20 millisecond thruster pulses in translational and rotational axis directions, with opposing thrusters fired as well so no net acceleration results.

A continuous fault check of the SAFER subsystems is performed by the monitoring function, comprising the following tests:

1. Leak monitoring
2. Battery voltage checks
3. Tank pressure and temperature checks
4. Regulator pressure and temperature checks
5. Battery pack temperature check

Status information resulting from continuous monitoring is displayed on the HCM LCD during SAFER flight. The following items are displayed in order:

1. Default display, showing GN_2 and power percent remaining
2. Pressure and temperature
3. Rotation rate

X	Pitch	Yaw	Always turned on	On if no roll command
−	−	−	B4	B2 B3
−	−		B3 B4	
−	−	+	B3	B1 B4
−		−	B2 B4	
−			B1 B4	B2 B3
−		+	B1 B3	
−	+	−	B2	B1 B4
−	+		B1 B2	
−	+	+	B1	B2 B3
	−	−	B4 F1	
	−		B4 F2	
	−	+	B3 F2	
		−	B2 F1	
		+	B3 F4	
	+	−	B2 F3	
	+		B1 F3	
	+	+	B1 F4	
+	−	−	F1	F2 F3
+	−		F1 F2	
+	−	+	F2	F1 F4
+		−	F1 F3	
+			F2 F3	F1 F4
+		+	F2 F4	
+	+	−	F3	F1 F4
+	+		F3 F4	
+	+	+	F4	F2 F3

Table C.2: Thruster select logic for X, pitch, and yaw commands.

Y	Z	Roll		Always turned on	On if no pitch or yaw
−	−	−	NA		
−	−		NA		
−	−	+	NA		
−		−		L1R	L1F L3F
−				L1R L3R	L1F L3F
−		+		L3R	L1F L3F
−	+	−	NA		
−	+		NA		
−	+	+	NA		
	−	−		U3R	U3F U4F
	−			U3R U4R	U3F U4F
	−	+		U4R	U3F U4F
		−		L1R R4R	
		+		R2R L3R	
	+	−		D2R	D1F D2F
	+			D1R D2R	D1F D2F
	+	+		D1R	D1F D2F
+	−	−	NA		
+	−		NA		
+	−	+	NA		
+		−		R4R	R2F R4F
+				R2R R4R	R2F R4F
+		+		R2R	R2F R4F
+	+	−	NA		
+	+		NA		
+	+	+	NA		

Table C.3: Thruster select logic for Y, Z, and roll commands.

4. Angular displacement
5. Battery voltage
6. High rate recorder status
7. Message display (error queue)

The fault detection system also provides for ground checkout of the SAFER flight unit. This function processes commands for data requests or avionics tests from ground support equipment connected to the CEA's ground servicing serial port.

C.2 SAFER EVA Flight Operation Requirements

The full SAFER system has requirements that cover flight operations as well as support procedures such as pre-EVA checkout, propellant recharging, and battery pack changing. Our SAFER example focuses on a subset of the full requirements, namely, those covering flight operations during an EVA. Furthermore, several requirements are incorporated in modified form to better suit the purposes of the example. The most significant change is that the controller samples switches and sensors on every frame rather than accepting change notifications via a serial line interface. This leads to the conceptually simpler architecture of a pure sampled-data control system.

C.2.1 Hand Controller Module (HCM)

The HCM provides the controls and displays for the SAFER crewmember to operate SAFER and to monitor status.

(1) The HCM shall comprise two units, the Hand Controller Unit (HCU) and the Display and Control Unit (DCU).

(2) The HCM shall provide the controls and displays for the SAFER crewmember to operate SAFER and to monitor status.

C.2.1.1 Display and Control Unit (DCU)

The DCU provides displays to the crew and switches for crew commands to power the SAFER, to select modes, and to select displays.

(3) The DCU shall provide a red LED and shall illuminate it whenever an electrical on-command is applied to any one of the SAFER thrusters.

(4) The DCU shall provide a green LED and shall illuminate it whenever automatic attitude hold (AAH) is enabled for one or more SAFER rotational axes.

(5) The DCU shall provide a 16-character, backlit liquid crystal display (LCD).

(6) The DCU shall display SAFER instructions and status information to the SAFER crewmember on the LCD.

(7) The DCU shall provide a three-position toggle switch to power the SAFER unit on and to control the SAFER test functions.

(8) The power toggle switch shall be oriented towards the crewmember for “TST,” in the center position for “ON,” and away for “OFF.”

(9) The DCU shall provide a three-position, momentary toggle switch to control the LCD display.

(10) The display toggle switch shall remain in the center position when not being used and shall be oriented so that positioning the switch towards or away from the crewmember will control the LCD menu selection.

(11) The DCU shall provide a two-position toggle switch to be used to direct hand controller commands for either full rotation or full translation control mode.

(12) The mode select toggle switch shall be positioned to the crewmember’s left for the Rotation Mode and to the crewmember’s right for the Translation Mode.

C.2.1.2 Hand Controller Unit (HCU)

The HCU provides those functions associated with the hand controller and the automatic attitude hold (AAH) pushbutton switch.

(13) The HCU shall provide a four-axis hand controller having three rotary axes and one transverse axis, operated by a side-mounted hand grip as depicted in Figure C.4.

(14) The HCU shall indicate primary control motions when the grip is deflected from the center or null position to mechanical hard-stops.

(15) The grip deflections shall result in six degree-of-freedom commands related to control axes as depicted in Figures C.5 and C.6.

(16) The HCU shall terminate commands when the grip is returned to the null position.

(17) The HCU shall provide a pushbutton switch to activate and deactivate AAH.

(18) The pushbutton switch shall activate AAH when depressed a single time.

(19) The pushbutton switch shall deactivate AAH when pushed twice within 0.5 second.

C.2.2 Propulsion Subsystem

SAFER thrusters are actuated by the control electronics assembly (CEA) using the valve drive assemblies (VDAs).

(20) The propulsion subsystem shall provide 24 gaseous nitrogen (GN_2) thrusters arranged as shown in Figure C.3.

(21) The VDAs shall accept thruster firing commands from the CEA and apply appropriate voltages to the selected thrusters.

(22) The VDAs shall have the capability of sensing current flow through the thruster solenoids and providing discrete signals to the CEA indicating such an event.

(23) The propulsion subsystem shall provide two transducers to monitor tank pressure and regulator outlet pressure.

(24) The propulsion subsystem shall provide two temperature sensors to measure tank temperature and regulator outlet temperature.

C.2.3 Avionics Assemblies

The avionics subsystem includes several assemblies housed within the backpack propulsion module, each having a digital interface to the CEA.

C.2.3.1 Inertial Reference Unit (IRU)

(25) The IRU shall provide angular rate sensors and associated electronics to measure rotation rates in each angular axis (roll, pitch, yaw).

(26) The IRU shall provide a temperature sensor for each angular rate sensor to allow temperature-based compensation.

(27) The IRU shall provide accelerometers to measure linear accelerations in each translation axis (X, Y, Z).

C.2.3.2 Power Supply Assembly (PSA)

(28) The power supply shall provide a voltage sensor to measure the valve drive battery voltage.

(29) The power supply shall provide a voltage sensor to measure the electronics battery voltage.

(30) The power supply shall provide a temperature sensor to measure battery pack temperature.

C.2.3.3 Data Recorder Assembly (DRA)

(31) The DRA shall accept performance data and system parameters from the CEA for storage and post-flight analysis.

(32) The DRA shall write formatted data on nonvolatile memory devices.

C.2.4 Avionics Software

Executing on a microprocessor within the control electronics assembly (CEA), the SAFER avionics software provides the capability to control SAFER flight maneuvers, to check out functionality and detect faults in SAFER, and to display SAFER fault conditions and general health and consumable status.

(33) The avionics software shall reference all commands and maneuvers to the coordinate system defined in Figure C.3.

(34) The avionics software shall provide a six degree-of-freedom maneuvering control capability in response to crewmember-initiated commands from the hand controller module.

(35) The avionics software shall allow a crewmember with a single command to cause the measured SAFER rotation rates to be reduced to less than 0.3 degree per second in each of the three rotational axes.

(36) The avionics software shall attempt to maintain the calculated attitude within ±5 degrees of the attitude at the time the measured rates were reduced to the 0.3 degree per second limit.

(37) The avionics software shall disable AAH on an axis if a crewmember rotation command is issued for that axis while AAH is active.

(38) Any hand controller rotation command present at the time AAH is initiated shall subsequently be ignored until a return to the off condition is detected for that axis or until AAH is disabled.

(39) Hand controller rotation commands shall suppress any translation commands that are present, but AAH-generated rotation commands may coexist with translations.

(40) At most one translation command shall be acted upon, with the axis chosen in priority order X, Y, Z.

(41) The avionics software shall provide accelerations with a maximum of four simultaneous thruster firing commands.

(42) The avionics software shall select thrusters in response to integrated AAH and crew-generated commands according to Tables C.2 and C.3.

(43) The avionics software shall provide flight control for AAH using the IRU-measured rotation rates and rate sensor temperatures.

(44) The avionics software shall provide fault detection for propulsion subsystem leakage in excess of 0.3% of GN_2 mass per second while thrusters are not firing.

(45) The avionics software shall provide limit checks for battery temperature and voltages, propulsion tank pressure and temperature, and regulator pressure and temperature.

C.2.5 Avionics Software Interfaces

The avionics software accepts input data from SAFER components by sampling the state of switches and digitized sensor readings. Outputs provided by the avionics software to SAFER components are transmitted in a device-specific manner.

(46) The avionics software shall accept the following data from the hand controller module:

- + pitch, – pitch
- + X, – X
- + yaw or + Y, – yaw or – Y
- + roll or + Z, – roll or – Z
- Power/test switch
- Mode switch
- Display proceed switch
- AAH pushbutton

(47) The avionics software shall accept the following data from the propulsion subsystem:

- Tank pressure and temperature
- Regulator pressure and temperature
- Thruster-on signal

(48) The avionics software shall accept the following data from the inertial reference unit:

- Roll, pitch, and yaw rotation rates
- Roll, pitch, and yaw sensor temperatures
- X, Y, and Z linear accelerations

(49) The avionics software shall accept the following data from the power supply:

- Valve drive battery voltage
- Electronics battery voltage
- Battery pack temperature

(50) The avionics software shall provide the following data to the HCM for display:

- Pressure, temperature, and voltage measurements
- Alert indications
- Rotation rates and displacements
- Crew prompts
- Failure messages
- Miscellaneous status messages

(51) The avionics software shall provide the following data to the valve drive assemblies for each of the 24 thrusters:

- Thruster on/off indications

(52) The avionics software shall provide the following data to the data recorder assembly:

- IRU-sensed rotation rates
- IRU-sensed linear accelerations
- IRU rate sensor temperatures
- Angular displacements
- AAH command status

C.3 Formalization of SAFER Requirements

A PVS formalization of the SAFER system described thus far is presented below[2]. A subset of the SAFER requirements has been chosen for modeling that emphasizes the main functional requirements and omits support functions such as the ground checkout features. Even within the flight operation requirements some functions have been represented only in abstract form.

---

[2] The PVS source files for the SAFER example are available on LaRC's Web server in the directory ftp://atb-www.larc.nasa.gov/Guidebooks/

C.3.1 PVS Language Features

Only a few PVS language features need to be understood to read the formal specification that follows. PVS specifications are organized around the concept of *theories*. Each theory is composed of a sequence of declarations or definitions of various kinds. Definitions from other theories are not visible unless explicitly imported.

PVS allows the usual range of scalar types to model various quantities. Numeric types include natural numbers (`nat`), integers (`int`), rationals (`rat`), and reals (`real`). Nonnumeric types include booleans (`bool`) and enumeration types (`{C1, C2, ...}`). Subranges and subtyping mechanisms allow derivative types to be introduced. Uninterpreted, nonempty type are of type `TYPE+`.

Structured data types or record types are used extensively in specifications. These types are introduced via declarations of the following form:

```
record_type: TYPE = [# v1: type_1, v2: type_2, ... #]
```

The first component of this record may be accessed using the notation `v1(R)`. A record value constructed from individual component values may be synthesized as follows:

```
(# v1 := <expression 1>, v2 := <expression 2>, ... #)
```

Similar to records are tuples, introduced via declarations of an analogous form:

```
tuple_type:  TYPE = [type_1, type_2, ... ]
```

The first component of a tuple may be accessed using the notation `proj_1(T)`. A tuple value constructed from individual component values may be synthesized as follows:

```
(<expression 1>, <expression 2>, ... )
```

An important class of types in PVS is formed by the function types. A declaration of the form:

```
fun_type: TYPE = [type_1 -> type_2]
```

defines a (higher-order) type whose values are functions from `type_1` to `type_2`. Function values may be constructed using lambda expressions:

```
(LAMBDA x, y: <expression of x, y>)
```

Logical variables are introduced to serve as arguments to functions and to express logical formulas or assertions:

```
x, y, z: VAR var_type
```

Local variable declarations also are available in most cases. Global variable declarations apply throughout the containing theory but no further.

A named function is defined quite simply by the following notation:

```
fn (arg_1, arg_2, ...): result_type =  <expression>
```

Each of the variables `arg_i` must have been declared of some type previously or given a local type declaration. The function definition must be mathematically well-defined, meaning its single result value is a function of the arguments and possibly some constants. No "free" variables are allowed within the expression. In addition, the type of the expression must be compatible with the result type.

Besides fully defining functions, it is possible to declare unspecified functions using the notation:

```
fn (arg_1, arg_2, ...): result_type
```

In this case, the function's signature is provided, but there is no definition. This is often useful when developing specifications in a top-down fashion. Also, it may be that some functions will never become defined in the specification, in which case they can never be expanded during a proof.

One type of expression in PVS is particularly useful for expressing complex functions. This feature, known as a LET expression, allows the introduction of bound variable names to refer to subexpressions.

```
LET v1 = <expression 1>, v2 = <expression 2>, ...
IN  <expression involving v1, v2, ...>
```

Each of the variables serves as a shorthand notation used in the final expression. The meaning is the same as if each of the subexpressions were substituted for its corresponding variable.

Finally, PVS provides a tabular notation for expressing conditional expressions in a naturally readable form. For example, an algebraic sign function could be defined as follows:

```
sign(x): signs = TABLE  %-------------------------%
                        |[ x < 0 | x = 0 | x > 0 ]|
                        %-------------------------%
                        |   -1   |   0   |   1    |
                        %-------------------------%
                 ENDTABLE
```

C.3.2 Overview of Formalization

The formal model uses a state machine representation of the main control function. The controller is assumed to run continuously, executing its control algorithms once per frame, whose duration is set at 5 milliseconds. In each frame, sensors, switches and the hand grip controller are sampled to provide the inputs to the control functions for that frame. Based on these inputs and the controller's state variables, actuator commands and crew display outputs are generated, as well as the controller's state for the next frame.

Eleven PVS theories are used to formalize the requirements:

- `avionics_types`
- `hand_controller_module`
- `propulsion_module`
- `inertial_reference_unit`
- `automatic_attitude_hold`
- `thruster_selection`
- `power_supply`
- `data_recorder`
- `self_test`
- `HCM_display`
- `avionics_model`

The full text of these theories is presented in Section C.3.3. The theories have been typechecked by PVS, and all resulting TCCs (type correctness conditions) have been proved.

Construction of the PVS specifications proceeded in a mostly top-down manner initially, but once parsing and typechecking of the PVS source was begun, the lower-level portions needed to be provided. For this reason, basic types tend to be done early during specification development and many of the definitions "meet in the middle" as both the upper and lower layers of the hierarchy are pushed toward completion. The following paragraphs point out some highlights of a subset of the theories, serving to annotate the PVS specifications of Section C.3.3.

C.3.2.1 Basic Types

A few common type definitions are provided for use elsewhere within the specification. Sensor readings are all modeled as real numbers. Several enumeration types are introduced to model translation and rotation commands. The `six_dof_command` type is a record that integrates all six axis commands. A few constants are also included to give names and values to null commands.

C.3.2.2 Hand Controller Module

The HCM switches are modeled in this theory as is the hand grip mechanism. The derivation of a six degree-of-freedom command from the four-axis hand controller based on current mode is defined here. Basic types for the LEDs and character display are included as well. A display buffer is modeled as an array of `character_display` values. A buffer pointer selects which element is currently being displayed. The pointer is updated when the previous state of the display proceed switch is neutral and the switch makes a transition in the up or down direction.

C.3.2.3 Propulsion Module

Thruster names are introduced via an enumeration type for the full complement of 24 thrusters. A more elaborate type called `thruster_desig` represents thruster designations in terms of their three component parts. This makes use of an advanced feature of the PVS language known as dependent types, where the type of later components of a record or tuple may depend on the value of earlier components. A mapping from names to designations is also provided. Finally, lists of thrusters are used to model actuator commands, where those thrusters to be fired are included in the list. Lists in PVS are analogous to the concept of lists in the Lisp programming language and its descendants.

C.3.2.4 Automatic Attitude Hold

A moderately complex part of the SAFER model revolves around the attitude hold feature. The hand grip pushbutton for engaging AAH mode is scanned to detect transitions that should be acted upon. The single-click, double-click engagement protocol is represented by the state diagram shown in Figure C.8, where the arcs are labeled with the switch values sensed in the current frame. The type `AAH_engage_state` denotes the states in this diagram, while the function `button_transition` models the diagram's transitions.

Several state components are modeled for managing AAH and its special requirements. The actual control law is not defined, but unspecified functions are provided to indicate where such processing fits in. The overall AAH transition function is defined by the PVS function `AAH_transition`, taking into account the conditions for activating and deactivating AAH on each axis, as well as the timeouts necessary for detecting double clicks of the AAH pushbutton.

C.3.2.5 Thruster Selection

Thruster selection takes place in two major steps: forming an integrated six degree-of-freedom command from the HCM command and AAH command, and then taking the integrated command and chosing individual thrusters to fire. Three functions take care of the first part by capturing the logic for prioritizing translation commands, merging rotation commands, and forming the correct aggregate command under the various conditions. Two functions called `BF_thrusters` and `LRUD_thrusters` formalize the thruster selection logic in Tables C.2 and C.3. These functions are defined using triply nested tables to avoid problems with cumbersome TCCs (type correctness conditions). A more readable form of the tables using triples of the axis commands resulted in large and intractable TCCs so the less pleasing form was necessary. This is an example of the tradeoffs that must be made occasionally when working with formal specifications. Finally, the theory concludes with the functions `selected_thrusters` and `selected_actuators` that integrate the results of the preceding functions to produce the final list of chosen thrusters.

C.3.2.6 Avionics Model

This top-level theory pulls together all the separate portions of the formalization. The overall state machine model for the SAFER controller is captured in the form of type definitions for the inputs, outputs, and state values, as well as the main state transition function called `SAFER_control`. Note the use of a LET expression to define most of the separate pieces that are merged to form the final outputs and next-state components. An initial state constant is also provided in this theory.

C.3.3 Full Text of PVS Theories

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%  The following PVS theories comprise a formal model of a subset
%%  of the control system functional requirements for an EVA
%%  propulsion system.  This example is heavily based on NASA's
%%  Simplified Aid for EVA Rescue (SAFER), developed at the Johnson
%%  Space Center (JSC).  For pedagogical reasons, the requirements
%%  deviate somewhat from the actual SAFER system.  Furthermore, the
%%  SAFER system is still under development.  As a result, the model
%%  that follows does not necessarily reflect the actual SAFER
%%  requirements as maintained by JSC.
%%
%%  References:
%%    1. Simplified Aid for EVA Rescue (SAFER) Operations Manual.
%%       NASA report JSC-26283, Sept. 1994.
%%    2. Simplified Aid for EVA Rescue (SAFER) Flight Test Project,
%%       Flight Test Article Prime Item Development Specification.
%%       NASA report JSC-25552A, July 1994.
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
avionics_types: THEORY
BEGIN

pressure:       TYPE = real
temperature:    TYPE = real
voltage:        TYPE = real

angular_rate:   TYPE = real
linear_accel:   TYPE = real

axis_command:   TYPE = {NEG, ZERO, POS}

tran_axis:      TYPE = {X, Y, Z}
rot_axis:       TYPE = {roll, pitch, yaw}

tran_command:   TYPE = [tran_axis -> axis_command]
rot_command:    TYPE = [rot_axis  -> axis_command]

rot_predicate:     TYPE = [rot_axis -> bool]

six_dof_command:   TYPE = [# tran: tran_command, rot: rot_command #]

null_tran_command: tran_command = (LAMBDA (a: tran_axis): ZERO)
null_rot_command:  rot_command  = (LAMBDA (a: rot_axis):  ZERO)

null_six_dof:   six_dof_command = (# tran := null_tran_command,
                                     rot  := null_rot_command #)

END avionics_types

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%  The hand controller module (HCM) consists of a set of switches,
%%  a hand grip controller with integral pushbutton, and a set of
%%  crew displays.  A six degree-of-freedom command is derived from
%%  four-axis hand grip inputs and the control mode switch position.
%%  It is assumed that switch debouncing is provided by a low-level
%%  hardware or software mechanism so that switch transitions in this
%%  model may be considered clean.
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
hand_controller_module: THEORY
BEGIN

IMPORTING avionics_types

power_test_switch:        TYPE = {OFF, ON, TST}

display_proceed_switch:   TYPE = {PREV, CURR, NEXT}

control_mode_switch:      TYPE = {ROT, TRAN}

AAH_control_button:       TYPE = {button_up, button_down}

HCM_switch_positions: TYPE = [#
        PWR:  power_test_switch,
        DISP: display_proceed_switch,
        MODE: control_mode_switch,
        AAH:  AAH_control_button
        #]


%%  The hand grip provides four axes for command input, which are
%%  multiplexed by the control mode switch into the required six axes.

hand_grip_position: TYPE =
        [# vert, horiz, trans, twist: axis_command #]

grip_command((grip: hand_grip_position),
             (mode: control_mode_switch)): six_dof_command =
        (# tran := null_tran_command WITH [
                     X := horiz(grip),
                     Y := IF mode = TRAN THEN trans(grip) ELSE ZERO ENDIF,
                     Z := IF mode = TRAN THEN vert(grip)  ELSE ZERO ENDIF],
           rot  := null_rot_command WITH [
                     roll  := IF mode = ROT THEN vert(grip)  ELSE ZERO ENDIF,
                     pitch := twist(grip),
                     yaw   := IF mode = ROT THEN trans(grip) ELSE ZERO ENDIF]
         #)


%%  The HCM display mechanism is centered around a 16-character LCD.
```

```
char_display_index: TYPE = {n: nat | 1 <= n & n <= 16} CONTAINING 1

character_display:  TYPE = [char_display_index -> character]

blank_char_display: character_display =
                      (LAMBDA (i: char_display_index): char(32))

HCM_display_set:    TYPE = [#
        LCD: character_display,
        THR: bool,
        AAH: bool
        #]

%%  Multiline messages are stored in a buffer and viewed one line
%%  at a time.

HCM_buffer_len:     above[0]    %% Any integer > 0

HCM_buffer_index:   TYPE = {n: nat | 1 <= n & n <= HCM_buffer_len}
                           CONTAINING 1

HCM_display_buffer: TYPE = [HCM_buffer_index -> character_display]

blank_display_buffer: HCM_display_buffer =
                        (LAMBDA (i: HCM_buffer_index): blank_char_display)

%%  The current pointer in the display state identifies which line to
%%  display, and the pointer can be moved up and down using the display
%%  proceed switch.

HCM_display_state:  TYPE = [#
        switch:  display_proceed_switch,
        buffer:  HCM_display_buffer,
        current: HCM_buffer_index
        #]

next_disp_pointer((new_sw:  display_proceed_switch),
                  (display: HCM_display_state)): HCM_buffer_index =
        IF switch(display) = CURR AND new_sw /= CURR
           THEN IF new_sw = PREV
                   THEN max(1, current(display) - 1)
                   ELSE min(HCM_buffer_len, current(display) + 1)
                ENDIF
           ELSE current(display)
        ENDIF

END hand_controller_module

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%  The propulsion module provides a number of sensors and a set of
%%  actuators to control the 24 thrusters, which are grouped into
%%  four clusters or quadrants.
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

propulsion_module: THEORY
BEGIN

IMPORTING avionics_types

propulsion_sensors: TYPE = [#
        tank_press:  pressure,
        tank_temp:   temperature,
        reg_press:   pressure,
        reg_temp:    temperature,
        thruster_on: bool
        #]


thruster_name:      TYPE =
        {B1, B2, B3, B4,  F1, F2, F3, F4,
         L1R, L1F,  R2R, R2F,  L3R, L3F,  R4R, R4F,
         D1R, D1F,  D2R, D2F,  U3R, U3F,  U4R, U4F}

%%  Thruster designators are triples of the form
%%    (thrust direction, cluster no., forward/rear location)
%%  Not all combinations of these values are possible so a dependent
%%  type is used to represent the constraints.

thruster_direction: TYPE = {UP, DN, BK, FD, LT, RT}
thruster_quadrant:  TYPE = {n: nat | 1 <= n & n <= 4} CONTAINING 1
thruster_location:  TYPE = {FW, RR}     % forward, rear
```

```
valid_quadrant((d: thruster_direction),
               (q: thruster_quadrant)): bool =
        COND d = UP -> q = 3 OR q = 4,
             d = DN -> q = 1 OR q = 2,
             d = LT -> q = 1 OR q = 3,
             d = RT -> q = 2 OR q = 4,
             ELSE   -> true
        ENDCOND

% Thrusters B1-B4 and F1-F4 are not normally written with a
% forward/rear location tag, but they are supplied below to fit
% the type declaration scheme.

thruster_desig:     TYPE = [
        dir: thruster_direction,
        {quad: thruster_quadrant | valid_quadrant(dir, quad)},
        {loc:  thruster_location |
                (dir = BK => loc = FW) AND (dir = FD => loc = RR)}
        ]


thruster_map(thruster: thruster_name): thruster_desig =
        TABLE thruster
             %--------------------%
             | B1  | (BK, 1, FW) ||
             | B2  | (BK, 2, FW) ||
             | B3  | (BK, 3, FW) ||
             | B4  | (BK, 4, FW) ||
             %--------------------%
             | F1  | (FD, 1, RR) ||
             | F2  | (FD, 2, RR) ||
             | F3  | (FD, 3, RR) ||
             | F4  | (FD, 4, RR) ||
             %--------------------%
             | L1R | (LT, 1, RR) ||
             | L1F | (LT, 1, FW) ||
             | R2R | (RT, 2, RR) ||
             | R2F | (RT, 2, FW) ||
             %--------------------%
             | L3R | (LT, 3, RR) ||
             | L3F | (LT, 3, FW) ||
                | R4R | (RT, 4, RR) ||
                | R4F | (RT, 4, FW) ||
               %--------------------%
                | D1R | (DN, 1, RR) ||
                | D1F | (DN, 1, FW) ||
                | D2R | (DN, 2, RR) ||
                | D2F | (DN, 2, FW) ||
               %--------------------%
                | U3R | (UP, 3, RR) ||
                | U3F | (UP, 3, FW) ||
                | U4R | (UP, 4, RR) ||
                | U4F | (UP, 4, FW) ||
               %--------------------%
        ENDTABLE


%%  Actuator commands are modeled as a list of thrusters to be fired.

thruster_list:      TYPE = list[thruster_name]

actuator_commands:  TYPE = thruster_list

null_actuation: actuator_commands = (: :)


END propulsion_module

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%  Sensing for angular rotation rates and linear acceleration is
%%  provided by the inertial reference unit (IRU).
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

inertial_reference_unit: THEORY
BEGIN

IMPORTING avionics_types

inertial_ref_sensors: TYPE = [#
        roll_rate:   angular_rate,
        pitch_rate:  angular_rate,
        yaw_rate:    angular_rate,
        roll_temp:   temperature,
        pitch_temp:  temperature,
        yaw_temp:    temperature,
        X_accel:     linear_accel,
        Y_accel:     linear_accel,
        Z_accel:     linear_accel
        #]

END inertial_reference_unit

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%  An automatic attitude hold (AAH) capability may be invoked to
%%  maintain near-zero rotation rates.  A pushbutton mounted on the
%%  hand grip engages AAH with a single button click, and disengages
%%  with a double click.  Internal state information is maintained
%%  to observe the button pushing protocol, keep track of status for
%%  each axis, and implement the attitude hold control law.
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

automatic_attitude_hold: THEORY
BEGIN

IMPORTING avionics_types, hand_controller_module,
          inertial_reference_unit, propulsion_module

click_timeout: nat = 100     %% At most 0.5 second between button
                             %% pushes for a double click.

AAH_engage_state:  TYPE = {AAH_off, AAH_started, AAH_on,
                           pressed_once, AAH_closing, pressed_twice}

AAH_control_law_state:  TYPE+

AAH_state:         TYPE = [# active_axes:  rot_predicate,
                             ignore_HCM:   rot_predicate,
                             toggle:       AAH_engage_state,
                             timeout:      nat,
                             control_vars: AAH_control_law_state #]
```

```
all_axes_off(active: rot_predicate): bool =
        (FORALL (a: rot_axis): NOT active(a))


%%  On each frame, the sampled value of the AAH engage button is
%%  checked to determine whether AAH is engaging or disengaging.
%%  This function implements the AAH engagement state diagram.

button_transition((state:   AAH_engage_state),
                  (button:  AAH_control_button),
                  (active:  rot_predicate),
                  (clock:   nat),
                  (timeout: nat)): AAH_engage_state =
        TABLE
               state       ,             button
                           %---------------------------------%
                           |[ button_down   | button_up     ]|
          %--------------------------------------------------%
          | AAH_off        |  AAH_started   | AAH_off       ||
          | AAH_started    |  AAH_started   | AAH_on        ||
          | AAH_on         |  pressed_once  | state_A       ||
          | pressed_once   |  pressed_once  | AAH_closing   ||
          | AAH_closing    |  pressed_twice | state_B       ||
          | pressed_twice  |  pressed_twice | AAH_off       ||
          %--------------------------------------------------%
        ENDTABLE
          WHERE state_A =
                  IF all_axes_off(active) THEN AAH_off ELSE AAH_on ENDIF,
                state_B =
                  IF all_axes_off(active) THEN AAH_off
                  ELSIF clock > timeout   THEN AAH_on ELSE AAH_closing
                  ENDIF


%%  The control law used to implement attitude hold is represented by two
%%  functions that map sensor inputs and control law state into next state
%%  and output values.

AAH_control_law((IRU_sensors:  inertial_ref_sensors),
                (prop_sensors: propulsion_sensors),
                (AAH_state:    AAH_state)): AAH_control_law_state
```

```
AAH_control_out((IRU_sensors:  inertial_ref_sensors),
                (prop_sensors: propulsion_sensors),
                (AAH_state:    AAH_state)): rot_command

initial_control_law_state: AAH_control_law_state


%%  AAH state information is updated in every frame.  Key transitions in
%%  the engage-state diagram cause various state variables to be set.

AAH_transition((IRU_sensors:  inertial_ref_sensors),
               (prop_sensors: propulsion_sensors),
               (button_pos:   AAH_control_button),
               (HCM_cmd:      six_dof_command),
               (AAH_state:    AAH_state),
               (clock:        nat)): AAH_state =
        LET engage = button_transition(toggle(AAH_state),
                                       button_pos,
                                       active_axes(AAH_state),
                                       clock,
                                       timeout(AAH_state)),
            starting = (toggle(AAH_state) = AAH_off AND engage = AAH_started)
        IN  (# active_axes  := (LAMBDA (a: rot_axis):
                                  starting OR
                                  (engage /= AAH_off AND
                                   active_axes(AAH_state)(a) AND
                                   (rot(HCM_cmd)(a) = ZERO OR
                                    ignore_HCM(AAH_state)(a)))),
               ignore_HCM   := (LAMBDA (a: rot_axis):
                                  IF starting
                                     THEN rot(HCM_cmd)(a) /= ZERO
                                     ELSE ignore_HCM(AAH_state)(a)
                                  ENDIF),
               toggle       := engage,
               timeout      := IF toggle(AAH_state) = AAH_on AND
                                  engage = pressed_once
                                  THEN clock + click_timeout
                                  ELSE timeout(AAH_state)
                               ENDIF,
               control_vars := AAH_control_law(IRU_sensors,
                                               prop_sensors,
                                               AAH_state)
            #)

END automatic_attitude_hold

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%  Thruster selection logic is formalized in the following theory.
%%  Hand controller and AAH commands are merged together in accordance
%%  with the various priority rules, yielding a six degree-of-freedom
%%  command.  Thruster selection tables are consulted to convert the
%%  translation and rotation components to individual actuator
%%  commands for opening suitable thruster valves.
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

thruster_selection: THEORY
BEGIN

IMPORTING avionics_types, propulsion_module, automatic_attitude_hold

rot_cmds_present(cmd: rot_command): bool =
        (EXISTS (a: rot_axis): cmd(a) /= ZERO)

%%  At most one translation is allowed, in priority order X, Y, Z.

prioritized_tran_cmd(tran: tran_command): tran_command =
        IF tran(X) /= ZERO
           THEN null_tran_command WITH [X := tran(X)]
        ELSIF tran(Y) /= ZERO
           THEN null_tran_command WITH [Y := tran(Y)]
        ELSIF tran(Z) /= ZERO
           THEN null_tran_command WITH [Z := tran(Z)]
           ELSE null_tran_command
        ENDIF

%%  Hand grip rotation commands take precedence over AAH commands
%%  unless inhibited at the start of AAH.

combined_rot_cmds((HCM_rot:    rot_command),
                  (AAH:        rot_command),
                  (ignore_HCM: rot_predicate)): rot_command =
        (LAMBDA (a: rot_axis):
                IF HCM_rot(a) = ZERO OR ignore_HCM(a)
                   THEN AAH(a)
                   ELSE HCM_rot(a)
                ENDIF)

%%  Hand grip rotations suppress translations but AAH rotations do not.

integrated_commands((HCM:   six_dof_command),
                    (AAH:   rot_command),
                    (state: AAH_state)): six_dof_command =
        IF all_axes_off(active_axes(state))
           THEN IF rot_cmds_present(rot(HCM))
                   THEN (# tran := null_tran_command,
                           rot  := rot(HCM) #)
                   ELSE (# tran := prioritized_tran_cmd(tran(HCM)),
                           rot  := null_rot_command #)
                ENDIF
           ELSE IF rot_cmds_present(rot(HCM))
                   THEN (# tran := null_tran_command,
                           rot  := combined_rot_cmds(rot(HCM), AAH,
                                                     ignore_HCM(state)) #)
                   ELSE (# tran := prioritized_tran_cmd(tran(HCM)),
                           rot  := AAH #)
                ENDIF
        ENDIF



%%  Selection of back and forward thrusters results in a pair of
%%  thrusters lists, the first of which gives mandatory thrusters
%%  and the second gives optional thrusters.  This function represents
%%  the selection table for X, pitch, and yaw commands.

thruster_list_pair: TYPE = [thruster_list, thruster_list]

BF_thrusters((A, B, C: axis_command)): thruster_list_pair =
        TABLE A
          | NEG  | TABLE B
                   | NEG  | TABLE C
                            %-------------------------------------%
                            | NEG  | ((: B4     :), (: B2, B3 :)) ||
                            | ZERO | ((: B3, B4 :), (:        :)) ||
                      | POS  | ((: B3     :), (: B1, B4 :)) ||
                      %--------------------------------------%
                    ENDTABLE ||
           | ZERO | TABLE C
                      %--------------------------------------%
                      | NEG  | ((: B2, B4 :), (:        :)) ||
                      | ZERO | ((: B1, B4 :), (: B2, B3 :)) ||
                      | POS  | ((: B1, B3 :), (:        :)) ||
                      %--------------------------------------%
                    ENDTABLE ||
           | POS  | TABLE C
                      %--------------------------------------%
                      | NEG  | ((: B2     :), (: B1, B4 :)) ||
                      | ZERO | ((: B1, B2 :), (:        :)) ||
                      | POS  | ((: B1     :), (: B2, B3 :)) ||
                      %--------------------------------------%
                    ENDTABLE ||
         ENDTABLE ||
| ZERO | TABLE B
           | NEG  | TABLE C
                      %--------------------------------------%
                      | NEG  | ((: B4, F1 :), (:        :)) ||
                      | ZERO | ((: B4, F2 :), (:        :)) ||
                      | POS  | ((: B3, F2 :), (:        :)) ||
                      %--------------------------------------%
                    ENDTABLE ||
           | ZERO | TABLE C
                      %--------------------------------------%
                      | NEG  | ((: B2, F1 :), (:        :)) ||
                      | ZERO | ((:        :), (:        :)) ||
                      | POS  | ((: B3, F4 :), (:        :)) ||
                      %--------------------------------------%
                    ENDTABLE ||
           | POS  | TABLE C
                      %--------------------------------------%
                      | NEG  | ((: B2, F3 :), (:        :)) ||
                      | ZERO | ((: B1, F3 :), (:        :)) ||
                      | POS  | ((: B1, F4 :), (:        :)) ||
                      %--------------------------------------%
                    ENDTABLE ||
         ENDTABLE ||
| POS  | TABLE B
```

```
                     | NEG  | TABLE C
                                %--------------------------------------%
                                | NEG  | ((: F1     :), (: F2, F3 :)) ||
                                | ZERO | ((: F1, F2 :), (:        :)) ||
                                | POS  | ((: F2     :), (: F1, F4 :)) ||
                                %--------------------------------------%
                              ENDTABLE ||
                     | ZERO | TABLE C
                                %--------------------------------------%
                                | NEG  | ((: F1, F3 :), (:        :)) ||
                                | ZERO | ((: F2, F3 :), (: F1, F4 :)) ||
                                | POS  | ((: F2, F4 :), (:        :)) ||
                                %--------------------------------------%
                              ENDTABLE ||
                     | POS  | TABLE C
                                %--------------------------------------%
                                | NEG  | ((: F3     :), (: F1, F4 :)) ||
                                | ZERO | ((: F3, F4 :), (:        :)) ||
                                | POS  | ((: F4     :), (: F2, F3 :)) ||
                                %--------------------------------------%
                              ENDTABLE ||
                   ENDTABLE ||
        ENDTABLE


%%  Selection of left, right, up, and down thrusters resulting from
%%  Y, Z, and roll commands.

LRUD_thrusters((A, B, C: axis_command)): thruster_list_pair =
        TABLE A
          | NEG  | TABLE B
                     | NEG  | TABLE C
                                %------------------------------------------%
                                | NEG  | ((:          :), (:          :)) ||
                                | ZERO | ((:          :), (:          :)) ||
                                | POS  | ((:          :), (:          :)) ||
                                %------------------------------------------%
                              ENDTABLE ||
                     | ZERO | TABLE C
                                %------------------------------------------%
                                | NEG  | ((: L1R      :), (: L1F, L3F :)) ||
                                | ZERO | ((: L1R, L3R :), (: L1F, L3F :)) ||
                      | POS  | ((: L3R      :), (: L1F, L3F :)) ||
                      %------------------------------------------%
                    ENDTABLE ||
           | POS  | TABLE C
                      %------------------------------------------%
                      | NEG  | ((:          :), (:          :)) ||
                      | ZERO | ((:          :), (:          :)) ||
                      | POS  | ((:          :), (:          :)) ||
                      %------------------------------------------%
                    ENDTABLE ||
         ENDTABLE ||
| ZERO | TABLE B
           | NEG  | TABLE C
                      %------------------------------------------%
                      | NEG  | ((: U3R      :), (: U3F, U4F :)) ||
                      | ZERO | ((: U3R, U4R :), (: U3F, U4F :)) ||
                      | POS  | ((: U4R      :), (: U3F, U4F :)) ||
                      %------------------------------------------%
                    ENDTABLE ||
           | ZERO | TABLE C
                      %------------------------------------------%
                      | NEG  | ((: L1R, R4R :), (:          :)) ||
                      | ZERO | ((:          :), (:          :)) ||
                      | POS  | ((: R2R, L3R :), (:          :)) ||
                      %------------------------------------------%
                    ENDTABLE ||
           | POS  | TABLE C
                      %------------------------------------------%
                      | NEG  | ((: D2R      :), (: D1F, D2F :)) ||
                      | ZERO | ((: D1R, D2R :), (: D1F, D2F :)) ||
                      | POS  | ((: D1R      :), (: D1F, D2F :)) ||
                      %------------------------------------------%
                    ENDTABLE ||
         ENDTABLE ||
| POS  | TABLE B
           | NEG  | TABLE C
                      %------------------------------------------%
                      | NEG  | ((:          :), (:          :)) ||
                      | ZERO | ((:          :), (:          :)) ||
                      | POS  | ((:          :), (:          :)) ||
                      %------------------------------------------%
                    ENDTABLE ||
                   | ZERO | TABLE C
                            %-----------------------------------------%
                            | NEG  | ((: R4R      :), (: R2F, R4F :)) ||
                            | ZERO | ((: R2R, R4R :), (: R2F, R4F :)) ||
                            | POS  | ((: R2R      :), (: R2F, R4F :)) ||
                            %-----------------------------------------%
                          ENDTABLE ||
                   | POS  | TABLE C
                            %-----------------------------------------%
                            | NEG  | ((:          :), (:          :)) ||
                            | ZERO | ((:          :), (:          :)) ||
                            | POS  | ((:          :), (:          :)) ||
                            %-----------------------------------------%
                          ENDTABLE ||
                 ENDTABLE ||
        ENDTABLE


%%  An integrated six degree-of-freedom command is mapped into a vector
%%  of actuator commands.  Selection tables provide lists of thrusters
%%  and both mandatory and optional thrusters are included as appropriate.

selected_thrusters(cmd: six_dof_command): thruster_list =
        LET (BF_mandatory, BF_optional) =
              BF_thrusters(tran(cmd)(X), rot(cmd)(pitch), rot(cmd)(yaw)),
            (LRUD_mandatory, LRUD_optional) =
              LRUD_thrusters(tran(cmd)(Y), tran(cmd)(Z), rot(cmd)(roll)),
            BF_thr =   append(IF rot(cmd)(roll) = ZERO
                                 THEN BF_optional
                                 ELSE (: :)
                              ENDIF,
                              BF_mandatory),
            LRUD_thr = append(IF rot(cmd)(pitch) = ZERO AND
                                 rot(cmd)(yaw) = ZERO
                                 THEN LRUD_optional
                                 ELSE (: :)
                              ENDIF,
                              LRUD_mandatory)
        IN  append(BF_thr, LRUD_thr)


selected_actuators((HCM:   six_dof_command),
                   (AAH:   rot_command),
                   (state: AAH_state)): actuator_commands =
        selected_thrusters(integrated_commands(HCM, AAH, state))

END thruster_selection

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%  Several sensors are provided by the power supply to support
%%  the fault monitoring functions.
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

power_supply: THEORY
BEGIN

IMPORTING avionics_types

power_supply_sensors: TYPE = [#
        elect_batt: voltage,
        valve_batt: voltage,
        batt_temp:  temperature
        #]

END power_supply

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%  A data recorder module is provided to record SAFER performance
%%  data for later analysis.
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

data_recorder: THEORY
BEGIN

IMPORTING avionics_types, propulsion_module,
          inertial_reference_unit, power_supply,
          automatic_attitude_hold

data_recorder_packet: TYPE+
data_packet((prop_sensors:  propulsion_sensors),
            (IRU_sensors:   inertial_ref_sensors),
            (power_sensors: power_supply_sensors),
            (AAH_state:     AAH_state),
            (thrusters:     actuator_commands)): data_recorder_packet

END data_recorder

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%  Continuous fault monitoring and consumables monitoring is
%%  provided by the self-test function.
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

self_test: THEORY
BEGIN

IMPORTING avionics_types, propulsion_module,
          inertial_reference_unit, power_supply,
          automatic_attitude_hold

self_test_state:   TYPE+

initial_self_test_state:  self_test_state

%%  The monitoring function is provided by the following.

SAFER_monitoring((prop_sensors:  propulsion_sensors),
                 (IRU_sensors:   inertial_ref_sensors),
                 (power_sensors: power_supply_sensors),
                 (self_test:     self_test_state)): self_test_state

END self_test

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%  Data from the various SAFER modules is collected for crew display
%%  through the HCM character display.
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
HCM_display: THEORY
BEGIN

IMPORTING avionics_types, hand_controller_module, self_test

%%  The HCM display buffer is constructed and updated by the following.

display_buffer((self_test:     self_test_state),
               (HCM_display:   HCM_display_buffer)): HCM_display_buffer

initial_display_buffer: HCM_display_buffer

END HCM_display

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%  The top level state machine model of the controller is presented
%%  in the following theory.  A transition function describes the
%%  effects of the controller's actions during a single frame.  A
%%  5 msec frame period is assumed (200 Hz sampling rate).
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

avionics_model: THEORY
BEGIN

IMPORTING avionics_types, hand_controller_module,
          propulsion_module, thruster_selection,
          inertial_reference_unit, automatic_attitude_hold,
          data_recorder, power_supply, self_test, HCM_display


%%  Controller inputs from SAFER modules and components.

avionics_inputs: TYPE = [#
        HCM_switches:  HCM_switch_positions,
        grip_command:  hand_grip_position,
        prop_sensors:  propulsion_sensors,
        IRU_sensors:   inertial_ref_sensors,
        power_sensors: power_supply_sensors
        #]
```

```
%%  Controller outputs to SAFER modules and components.

avionics_outputs: TYPE = [#
        HCM_displays:   HCM_display_set,
        prop_actuators: actuator_commands,
        data_recorder:  data_recorder_packet
        #]

%%  Internal state variables maintained by the controller.

avionics_state:  TYPE = [#
        msg_display:  HCM_display_state,
        AAH_state:    AAH_state,
        clock:        nat,
        self_test:    self_test_state
        #]

avionics_result: TYPE = [# output: avionics_outputs,
                           state:  avionics_state #]


%%  The top level state machine transition function represents one
%%  frame of controller operation (once around the main control loop).

SAFER_control ((avionics_inputs: avionics_inputs),
               (avionics_state:  avionics_state)): avionics_result =

        LET switches      = HCM_switches(avionics_inputs),
            raw_grip      = grip_command(avionics_inputs),

            prop_sensors  = prop_sensors(avionics_inputs),
            IRU_sensors   = IRU_sensors(avionics_inputs),
            power_sensors = power_sensors(avionics_inputs),

            AAH_state     = AAH_state(avionics_state),
            AAH_active    = NOT all_axes_off(active_axes(AAH_state)),
            display       = msg_display(avionics_state),
            clock         = clock(avionics_state),
            self_test     = self_test(avionics_state),

            grip_cmd      = grip_command(raw_grip, MODE(switches)),
            AAH_cmd       = AAH_control_out(IRU_sensors, prop_sensors,
                                            AAH_state),

            thrusters     = selected_actuators(grip_cmd, AAH_cmd, AAH_state),

            monitoring    = SAFER_monitoring(prop_sensors, IRU_sensors,
                                             power_sensors, self_test),

            disp_window   = buffer(display)(current(display)),
            disp_buffer   = display_buffer(monitoring, buffer(display)),
            disp_pointer  = next_disp_pointer(DISP(switches), display)
        IN
            (# output := (# HCM_displays :=
                              (# LCD := disp_window,
                                 THR := thruster_on(prop_sensors),
                                 AAH := AAH_active #),
                            prop_actuators := thrusters,
                            data_recorder  :=
                              data_packet(prop_sensors, IRU_sensors,
                                          power_sensors, AAH_state,
                                          thrusters)
                          #),
               state  := (# msg_display :=
                             (# switch  := DISP(switches),
                                buffer  := disp_buffer,
                                current := disp_pointer #),
                            AAH_state :=
                              AAH_transition(IRU_sensors, prop_sensors,
                                             AAH(switches), grip_cmd,
                                             AAH_state, clock),
                            clock     := 1 + clock,
                            self_test := monitoring
                          #)
             #)


%%  The controller is assumed to be powered up into the following
%%  initial state.

initial_avionics_state:  avionics_state =
        (# msg_display := (# switch  := CURR,
                             buffer  := initial_display_buffer,
                             current := 1
```

```
                    #),
      AAH_state   := (# active_axes  := (LAMBDA (a: rot_axis): false),
                        ignore_HCM   := (LAMBDA (a: rot_axis): false),
                        toggle       := AAH_off,
                        timeout      := 0,
                        control_vars := initial_control_law_state
                    #),
      clock     := 0,
      self_test := initial_self_test_state
    #)

END avionics_model
```

C.4 Analysis of SAFER

Having produced a formalized version of the SAFER requirements, several types of rigorous analysis are possible. Precisely stated requirements models have consequences that can themselves be precisely stated. By expressing various properties of the system or selected subsystem behavior, it is possible to analyze requirements, in a limited way, for well-formedness and compliance with desired characteristics. Once expressed in this manner, it is further possible to formally prove that the properties follow from the definitions given in the requirements model.

C.4.1 Formulating System Properties

From a basic model of the SAFER controller, there are many possible aspects of system behavior one might wish to investigate or verify. Some aspects might result from higher-level requirements or desired system characteristics. Examples of such properties are as follows:

- When AAH is inactive and no hand grip commands are present there should be no thruster firings.
- SAFER should never fire more than four thrusters simultaneously.
- No two selected thrusters should oppose each other, that is, have cancelling thrust with respect to the center of mass.
- Once AAH is turned off for a rotational axis it remains off until a new AAH cycle is initiated.

Properties such as these identify behavior that designers expect or require the system to have if it is to satisfy their expectations. These properties must logically follow as consequences of the definitions contained in the system model. Thus, if a mistake was made in deriving the requirements or formalizing them, attempts to express and prove these properties will help detect the error. This approach then constitutes a rigorous method of analyzing requirements. It becomes possible to definitively answer questions about system behavior, reducing the chances of error from miscalculation, interpretation, or engineering judgment.

C.4.1.1 Formalization of the Maximum Thruster Property

To illustrate the process of formalizing system properties, it is instructive to take one of the suggested properties mentioned above and capture it formally using PVS. Let the Maximum Thruster Property be the requirement that SAFER should never fire more than four thrusters simultaneously. This condition was expressed as an explicit requirement in Section C.2. It can be shown that it follows as a direct consequence of the more detailed functional requirements.

Thruster selection is a function of the hand grip command and any AAH-generated commands. Tables C.2 and C.3 are used to choose appropriate thrusters based on which commands appear. Examining the tables, it can be seen that as many as four thrusters can be selected from each, resulting, at first glance, in as many as eight thrusters chosen from both. Clearly, some other conditions are needed to reduce the possibilities. Several restrictive conditions make some command combinations invalid. In addition, the table entries themselves are interrelated in ways that limit the thruster count for multiple commands. Taking these restrictions and the table structure into account, the four-thruster maximum can be derived.

Expressing the Maximum Thruster Property in PVS is straightforward:

```
FORALL (a_in: avionics_inputs), (a_st: avionics_state):
  length(prop_actuators(output(SAFER_control(a_in, a_st)))) <= 4
```

This formula asserts that for any input and state values, the outputs produced by the SAFER controller, which include the list of thrusters to fire in the current frame, obey the maximum thruster requirement. This is a strong statement because it applies to any output that can be generated by the model.

Section C.4.1.2 presents a PVS theory containing the desired property and supporting lemmas needed to prove it. The property appears at the end of the theory, expressed as the PVS theorem called `max_thrusters`. All the preceding lemmas in this theory are used to construct the proof of `max_thrusters`. Some lemmas were drafted specifically to decompose the overall proof into manageable pieces, thus representing intermediate steps. Other lemmas, however, express various facts about the problem domain that are useful in their own right and might find application in other proof efforts.

C.4.1.2 PVS Theory for Maximum Thruster Property

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%  Some properties of the SAFER controller are formalized in the
%%  following PVS theory.  The top level theorem, max_thrusters,
%%  asserts that for any input and current state values, the SAFER
%%  controller will issue no more four thruster firing commands.
%%  The theorems and lemmas stated below have all been proved using
%%  the PVS interactive proof checker.
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

SAFER_properties: THEORY
BEGIN

IMPORTING avionics_model

A,B,C:    VAR axis_command
tr:       VAR tran_command
HCM,cmd:  VAR six_dof_command
AAH:      VAR rot_command
state:    VAR AAH_state
thr,U,V:  VAR thruster_list
act:      VAR actuator_commands
BF,LRUD:  VAR thruster_list_pair


%%  A simple list property is needed to support thruster selection proofs.

length_append: LEMMA
        length(append(U, V)) = length(U) + length(V)


%%  Only one translation command can be accepted for thruster selection.

only_one_tran(tr): bool =
        (tr(X) /= ZERO IMPLIES tr(Y) = ZERO AND tr(Z) = ZERO)
           AND (tr(Y) /= ZERO IMPLIES tr(Z) = ZERO)

only_one_tran_pri: LEMMA
        only_one_tran(prioritized_tran_cmd(tr))

only_one_tran_int: LEMMA
```

```
        only_one_tran(tran(integrated_commands(HCM, AAH, state)))


%%  All categories of selected thrusters (BF vs. LRUD and mandatory
%%  vs. optional) are bounded in size by two, which follows directly
%%  from inspection of the tables.

max_thrusters_BF: LEMMA
        length(proj_1(BF_thrusters(A, B, C))) <= 2 AND
        length(proj_2(BF_thrusters(A, B, C))) <= 2

max_thrusters_LRUD: LEMMA
        length(proj_1(LRUD_thrusters(A, B, C))) <= 2 AND
        length(proj_2(LRUD_thrusters(A, B, C))) <= 2


%%  Absence of translation commands implies no optional thrusters
%%  will be selected.

no_opt_thr_BF: LEMMA
        tr(X) = ZERO IMPLIES length(proj_2(BF_thrusters(tr(X), B, C))) = 0

no_opt_thr_LRUD: LEMMA
        tr(Y) = ZERO AND tr(Z) = ZERO IMPLIES
          length(proj_2(LRUD_thrusters(tr(Y), tr(Z), C))) = 0


%%  Top level theorems establishing bounds on number of selected thrusters:

max_thrusters_sel: LEMMA
        only_one_tran(tran(cmd)) IMPLIES
          length(selected_thrusters(cmd)) <= 4

max_thrusters: THEOREM
        FORALL (a_in: avionics_inputs), (a_st: avionics_state):
          length(prop_actuators(output(SAFER_control(a_in, a_st)))) <= 4

END SAFER_properties
```

C.4.2 Proving System Properties

Merely expressing anticipated facts about a system model may be sufficient to flush out errors or lead to the discovery of other noteworthy issues. To obtain further benefit from the formalization, a proof may be performed to make a highly convincing case for the absence of undesirable system behavior. While informal proofs could suffice in many cases, fully formal proofs with mechanical assistance offer the highest degree of assurance. Carrying out proofs within a system such as PVS can yield very high confidence in any results established, subject to the assumptions made about the system environment during the modeling effort.

C.4.2.1 Proof Sketch of the Maximum Thruster Property

The argument for why four thrusters is the maximum is as follows. In both of the thruster selection tables, there can be at most two mandatory thrusters and at most two optional thrusters selected. Consider whether there is a translation command present for the X axis.

- **Case 1: No X command present.** Inspection of Table C.2 shows that there will be no optional thrusters selected in this case. Next consider whether there is a pitch or yaw command present.
  - **Case 1.1: No pitch or yaw commands.** Inspection of Table C.2 shows that no thrusters at all are selected in this case. At most four can come from the other table. Hence, the bound holds.
  - **Case 1.2: Pitch or yaw command present.** Table C.3 shows that no optional thrusters are chosen from this table. Hence only mandatory thrusters from each table are chosen, which number at most four.
- **Case 2: X command present.** Because only one translation command is allowed, it follows that no Y or Z command can appear. This, in turn, implies that no optional thrusters are chosen from Table C.3. Now consider whether there is a roll command.
  - **Case 2.1: No roll command.** Without a roll command, no thrusters at all result from Table C.3. Thus the bound holds.
  - **Case 2.2: Roll command present.** A roll command implies that Table C.2 yields no optional thrusters. This leaves only mandatory thrusters from each table, and the bound of four thrusters is upheld.

The foregoing proof sketch is the case analysis used to tackle the formal proof carried out using PVS.

In the theory `SAFER_properties` from Section C.4.1.2, `max_thrusters` is the top level theorem whose proof is based on the lemmas `max_thrusters_sel` and `only_one_tran_int`. Each of these lemmas is, in turn, proved in terms of other lemmas from this theory. `max_thrusters_sel` had the most complex proof of the group; its proof involved the case analysis outlined above.

Section C.4.2.2 shows a transcript from the proof of theorem `max_thrusters`. This proof contained only five steps, each of which requires the user to supply a prover command. The notation of PVS proofs is based on a *sequent* representation. A sequent is a stylized way of normalizing a logical formula that has a convenient structure with useful symmetries. In a sequent, a (numbered) list of antecedent formulas is meant to imply a (numbered) list of consequent formulas:

```
[-2] <antecedent 2>
[-1] <antecedent 1>
   |-------
 [1] <consequent 1>
 [2] <consequent 2>
```

The antecedents are considered to form a conjunction and the consequents form a disjunction. Every user-supplied prover command or inference rule causes one or more new sequents to be generated that moves the proof closer to completion.

In the proof of `max_thrusters`, the five steps are as follows:

1. **Rule:** `(skosimp*).` This rule merely eliminates the outer universal quantifiers (from the FORALL expression) and simplifies the result. This is a commonly used command at the start of many proofs.

2. **Rule:** `(expand "SAFER_control").` The cited function is expanded in place by this rule, with all actual arguments propagated to their proper place.

3. **Rule:** `(expand "selected_actuators").` Another case of function expansion is used here.

4. **Rule:** `(use "only_one_tran_int").` One of the lemmas from the containing theory is imported for later use. The lemma's variables are automatically instantiated with terms that appear to be useful, which is easy to do in this case.

5. **Rule:** `(forward-chain "max_thrusters_sel").` Forward chaining is the application of a lemma of the form `P => Q` when formula `P` appears in the antecedent list. In this case, the whole sequent is actually an instance of the cited lemma, so invoking the forward chain rule finishes off the proof immediately.

Proofs of the remaining lemmas were all carried out within PVS in a similar fashion. Most required only a few steps. The exception was `max_thrusters_sel`, which required a more elaborate proof because of the case analysis mentioned above. This proof contained around 40 steps, resulting from several case splits and the subsequent equality substitutions to use the facts generated by the case splitting.

C.4.2.2 PVS Proof of Maximum Thruster Property

```
max_thrusters :

  |-------
{1}   FORALL (a_in: avionics_inputs), (a_st: avionics_state):
        length(prop_actuators(output(SAFER_control(a_in, a_st)))) <= 4

Rule? (skosimp*)
Repeatedly Skolemizing and flattening,
this simplifies to:
max_thrusters :

  |-------
{1}   length(prop_actuators(output(SAFER_control(a_in!1, a_st!1)))) <= 4

Rule? (expand "SAFER_control" )
Expanding the definition of SAFER_control,
this simplifies to:
max_thrusters :

  |-------
{1}   length(selected_actuators(grip_command(grip_command(a_in!1),
                                             MODE
                                               (HCM_switches(a_in!1))),
                                AAH_control_out(IRU_sensors(a_in!1),
                                                prop_sensors(a_in!1),
                                                AAH_state(a_st!1)),
                                AAH_state(a_st!1)))
          <= 4

Rule? (expand "selected_actuators" )
Expanding the definition of selected_actuators,
this simplifies to:
max_thrusters :

  |-------
{1}   length
        (selected_thrusters
           (integrated_commands(grip_command(grip_command(a_in!1),
                                             MODE
                                               (HCM_switches(a_in!1))),
                                AAH_control_out(IRU_sensors(a_in!1),
                                                prop_sensors(a_in!1),
                                                AAH_state(a_st!1)),
                                AAH_state(a_st!1))))
          <= 4

Rule? (use "only_one_tran_int")
Using lemma only_one_tran_int,
this simplifies to:
max_thrusters :

{-1}   only_one_tran(tran(integrated_commands(grip_command(grip_command(a_in!1),
                                                          MODE
                                                          (HCM_switches(a_in!1))),
                                             AAH_control_out
                                             (IRU_sensors(a_in!1),
                                              prop_sensors(a_in!1),
                                              AAH_state(a_st!1)),
                                             AAH_state(a_st!1))))
  |-------
[1]   length
        (selected_thrusters
           (integrated_commands(grip_command(grip_command(a_in!1),
                                             MODE
                                               (HCM_switches(a_in!1))),
                                AAH_control_out(IRU_sensors(a_in!1),
                                                prop_sensors(a_in!1),
                                                AAH_state(a_st!1)),
                                AAH_state(a_st!1))))
          <= 4

Rule? (forward-chain "max_thrusters_sel")
Forward chaining on max_thrusters_sel,
Q.E.D.

Run time  = 4.03 secs.
Real time = 73.92 secs.
```

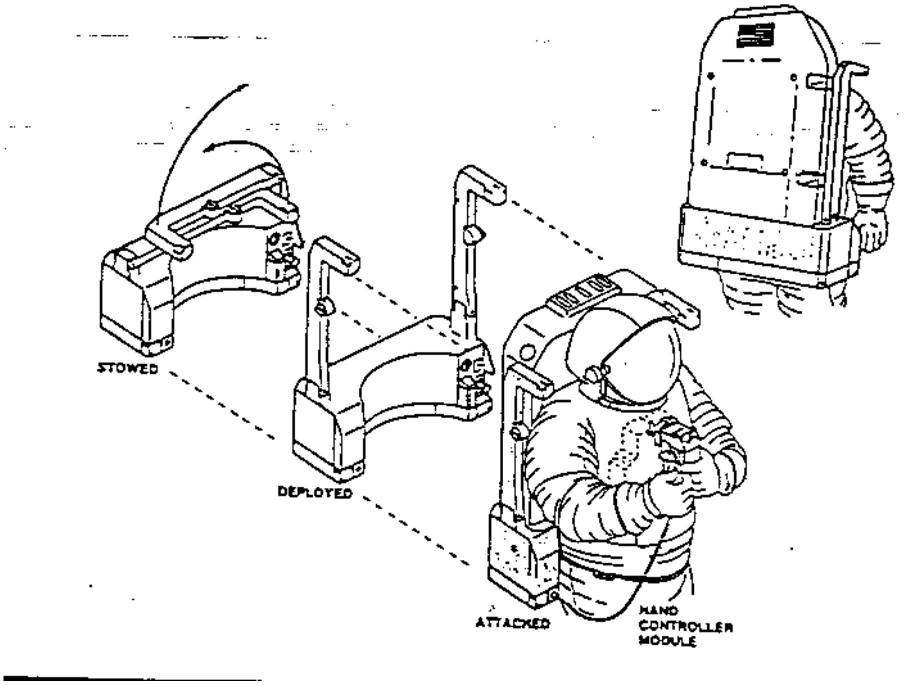

STOWED
DEPLOYED
ATTACHED
HAND CONTROLLER MODULE

Figure C.1: SAFER use by an EVA crewmember.

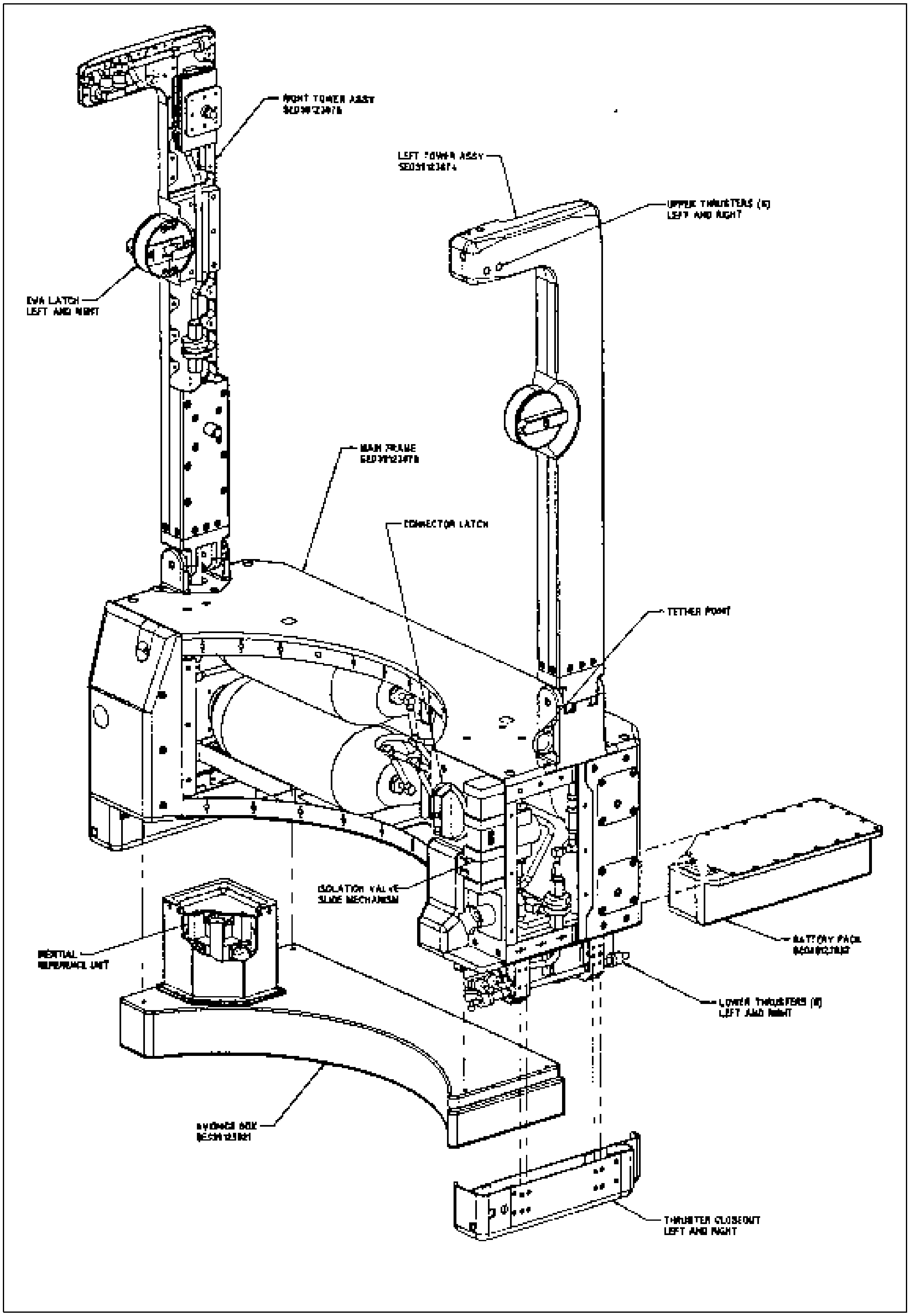
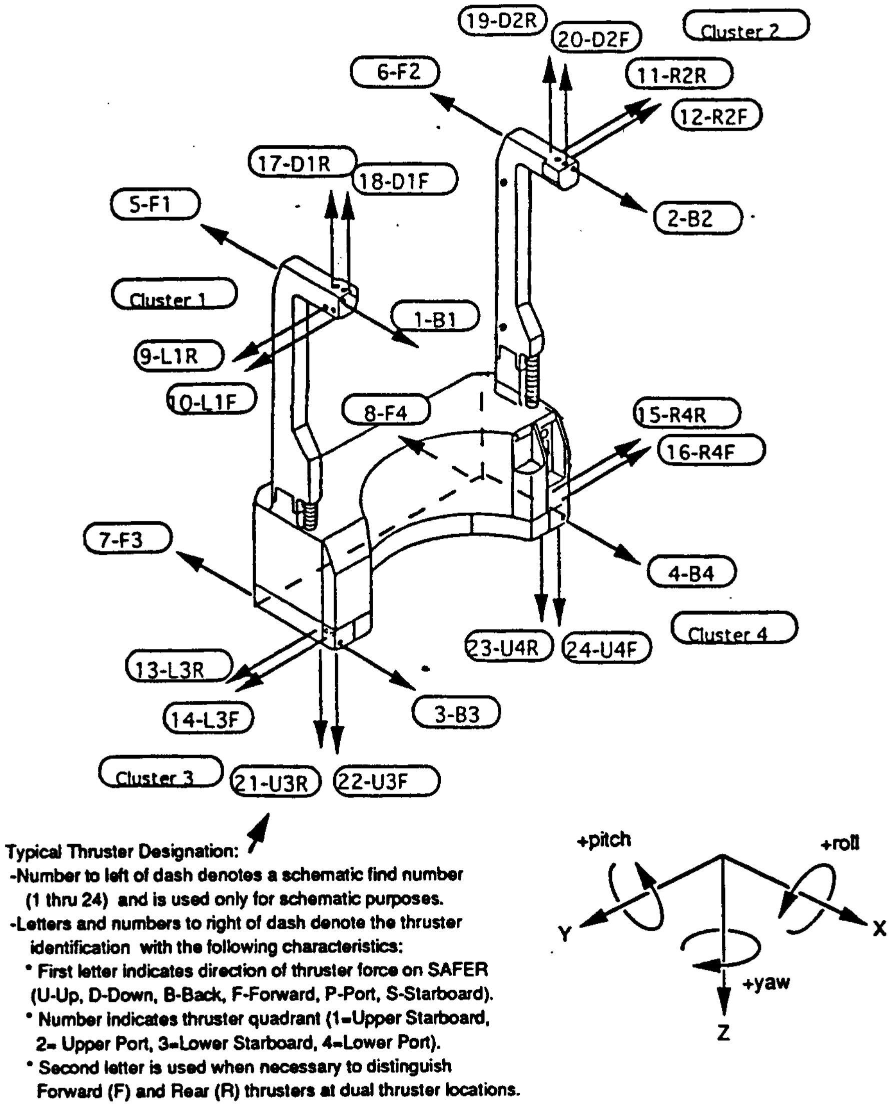

RIGHT TOWER ASSY
LEFT TOWER ASSY
UPPER THRUSTERS (8) LEFT AND RIGHT
EVA LATCH LEFT AND RIGHT
MAIN FRAME
CONNECTOR LATCH
TETHER POINT
ISOLATION VALVE SLIDE MECHANISM
INERTIAL REFERENCE UNIT
BATTERY PACK
LOWER THRUSTERS (8) LEFT AND RIGHT
AVIONICS BOX
THRUSTER CLOSEOUT LEFT AND RIGHT

Figure C.2: Propulsion module structure and mechanisms.


19-D2R
20-D2F
Cluster 2
6-F2
11-R2R
12-R2F
17-D1R
18-D1F
5-F1
2-B2
Cluster 1
1-B1
9-L1R
10-L1F
8-F4
15-R4R
16-R4F
7-F3
4-B4
Cluster 4
23-U4R
24-U4F
13-L3R
14-L3F
3-B3
Cluster 3
21-U3R
22-U3F
+pitch
+roll
Y
X
+yaw
Z

Typical Thruster Designation:

-Number to left of dash denotes a schematic find number (1 thru 24) and is used only for schematic purposes.

-Letters and numbers to right of dash denote the thruster identification with the following characteristics:

- First letter indicates direction of thruster force on SAFER (U-Up, D-Down, B-Back, F-Forward, P-Port, S-Starboard).
- Number indicates thruster quadrant (1=Upper Starboard, 2= Upper Port, 3=Lower Starboard, 4=Lower Port).
- Second letter is used when necessary to distinguish Forward (F) and Rear (R) thrusters at dual thruster locations.

Figure C.3: SAFER thrusters and axes.

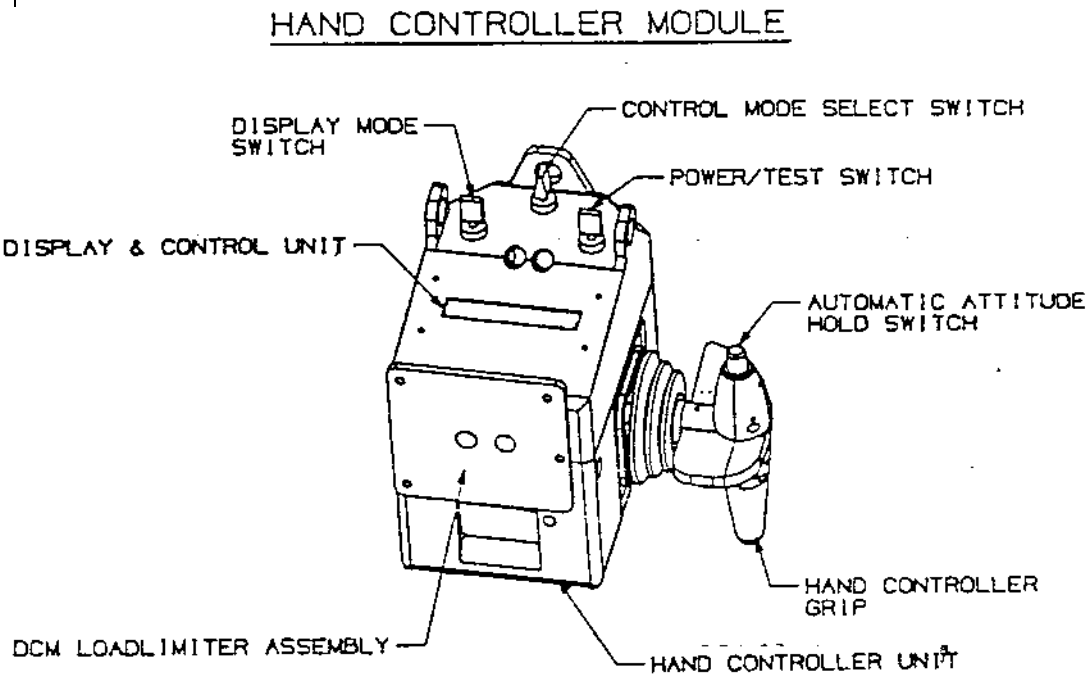

HAND CONTROLLER MODULE
DISPLAY MODE SWITCH
CONTROL MODE SELECT SWITCH
POWER/TEST SWITCH
DISPLAY & CONTROL UNIT
AUTOMATIC ATTITUDE HOLD SWITCH
HAND CONTROLLER GRIP
DCM LOADLIMITER ASSEMBLY
HAND CONTROLLER UNIT

Figure C.4: Hand controller module.

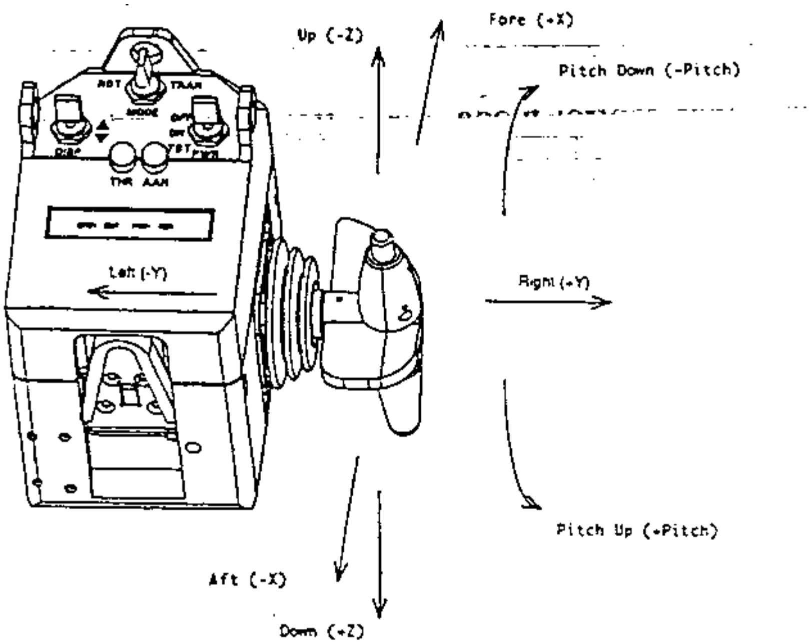

Up (-Z)
Fore (+X)
Pitch Down (-Pitch)
ROT
TRAN
MODE
OFF
ON
TST
PWR
DISP
THR
AAH
Left (-Y)
Right (+Y)
Pitch Up (+Pitch)
Aft (-X)
Down (+Z)

Figure C.5: Hand controller translational axes.

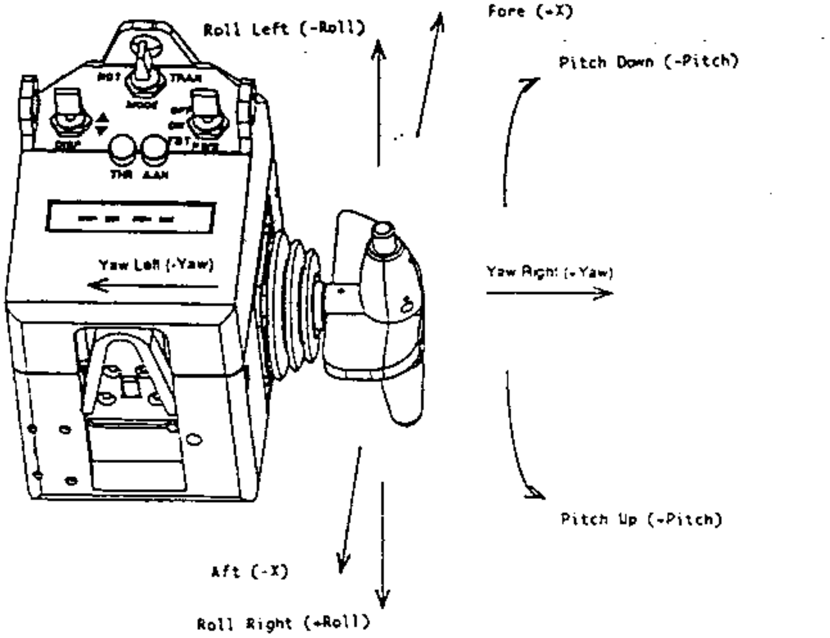

Fore (+X)
Roll Left (-Roll)
Pitch Down (-Pitch)
ROT
TRAN
MODE
THR
Yaw Left (-Yaw)
Yaw Right (+Yaw)
Pitch Up (+Pitch)
Aft (-X)
Roll Right (+Roll)

Figure C.6: Hand controller rotational axes.

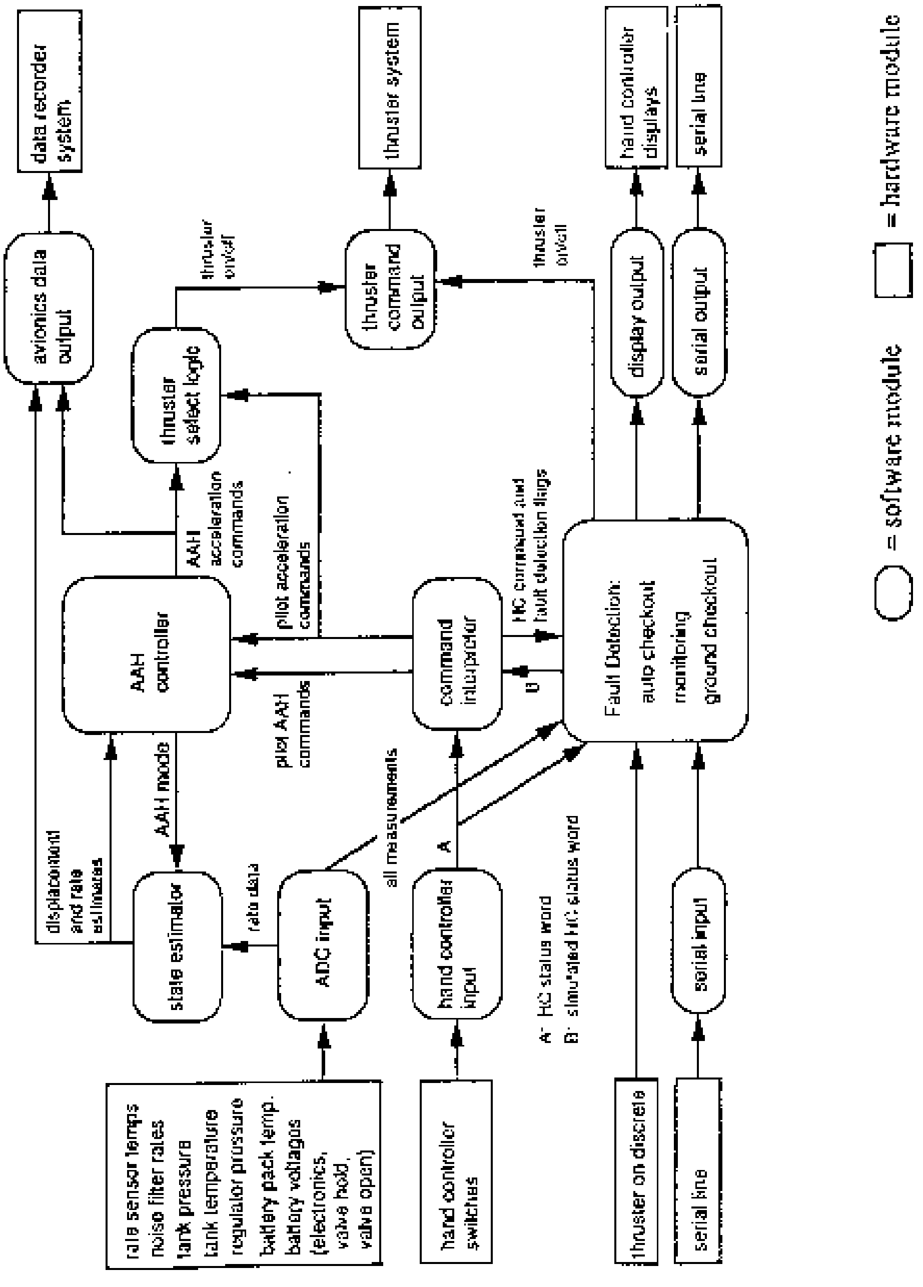

data recorder system
avionics data output
thruster select logic
thruster on/off
thruster system
thruster command output
thruster on/off
hand controller displays
serial line
display output
serial output
= hardware module
= software module
Fault Detection:
auto checkout
monitoring
ground checkout
HC command and fault detection flags
B
command interpreter
pilot acceleration commands
AAH acceleration commands
AAH controller
pilot AAH commands
AAH mode
displacement and rate estimates
state estimator
rate data
ADC input
all measurements
A
hand controller input
A: HC status word
B: simulated HC status word
serial input
thruster on discrete
serial line
hand controller switches
rate sensor temps
noise filter rates
tank pressure
tank temperature
regulator pressure
battery pack temp.
battery voltages
(electronics,
valve hold,
valve open)

Figure C.7: SAFER system software architecture.

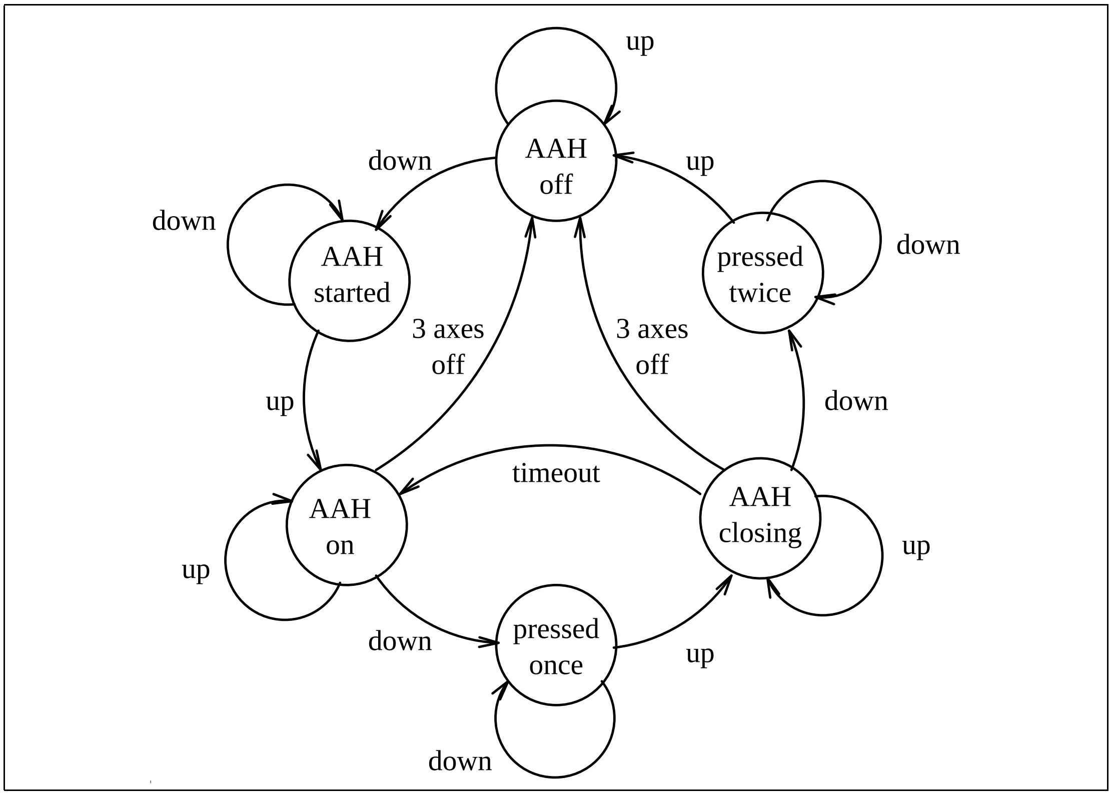

up
AAH off
down
up
down
AAH started
pressed twice
down
3 axes off
3 axes off
up
down
timeout
AAH on
AAH closing
up
up
down
pressed once
up
down

Figure C.8: AAH pushbutton state diagram.